OBJECT ORIENTED ANALYSIS & DESIGN USING UML

UNIT-I:
Introduction: The Structure of Complex systems, The Inherent Complexity of Software,
Attributes of Complex System, Organized and Disorganized Complexity, Bringing Order to
Chaos, Designing Complex Systems, Evolution of Object Model, Foundation of Object Model,
Elements of Object Model, Applying the Object Model.

UNIT-II:
Classes and Objects: Nature of object, Relationships among objects, Nature of a Class,
Relationship among Classes, Interplay of Classes and Objects, Identifying Classes and Objects,
Importance of Proper Classification, Identifying Classes and Objects, Key abstractions and
Mechanisms.

UNIT-III:
Introduction to UML: Why we model, Conceptual model of UML, Architecture, Classes,
Relationships, Common Mechanisms, Class diagrams, Object diagrams.

The Inherent Complexity of Software

The Properties of Simple and Complex Software Systems

A dying star on the verge of collapse, a child learning how to read, white blood cells rushing to attack a virus: these are but a few of the objects in the physical world that involve truly awesome complexity. Software may also involve elements of great complexity; however, the complexity we find here is of a fundamentally different kind. As Brooks points out, "Einstein argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity" .

We do realize that some software systems are not complex. These are the largely forgettable applications that are specified, constructed, maintained, and used by the same person, usually the amateur programmer or the professional developer working in isolation. This is not to say that all such systems are crude and inelegant, nor do we mean to belittle their creators. Such systems tend to have a very limited purpose and a very short life span. We can afford to throw them away and replace them with entirely new software rather than attempt to reuse them, repair them, or extend their functionality, Such applications are generally more tedious than difficult to develop; consequently, learning how to design them does not interest us.Instead, we are much more interested in the challenges of developing what we will call *industrial-strength software*. Here we find applications that exhibit a very rich set of behaviors, as, for example, in reactive systems that drive or are driven by events in the physical world, and for which time and space are scarce resources; applications that maintain the integrity of hundreds of thousands of records of information while allowing concurrent updates and queries; and systems for the command and control of real-world entities, such as the routing of air or railway traffic. Software systems such as these tend to have a long life span, and over time, many users come to depend upon their proper functioning. In the world of industrial-strength software, we also find frameworks that simplify the creation of domain-specific applications, and programs that mimic some aspect of human intelligence. Although such applications are generally products of research and development they are no less complex, for they are the means and artifacts of incremental and exploratory development.

The distinguishing characteristic of industrial-strength software is that it is intensely difficult, if not impossible, for the individual developer to comprehend all the subtleties of its design. Stated in blunt terms, the complexity of such systems

exceeds the human intellectual capacity. Alas, this complexity we speak of seems to be an essential property of all large software systems. By *essential* we mean that we may master this complexity, but we can never make it go away.

Certainly, there will always be geniuses among us, people of extraordinary skill who can do the work of a handful of mere mortal developers, the software engineering equivalents of Frank Lloyd Wright or Leonardo da Vinci. These are the people whom we seek to deploy as our systems architects: the ones who devise innovative idioms, mechanisms, and frameworks that others can use as the architectural foundations of other applications or systems. However, as Peters observes, "The world is only sparsely populated with geniuses. There is no reason to believe that the software engineering community has an inordinately large proportion of then" [2]. Although there is a touch of genius in all of us, in the realm of industrial-strength software we cannot always rely upon divine inspiration to carry us through. Therefore, we must consider more disciplined ways to master complexity. To better understand what we seek to control, let us next examine why complexity is an essential property of all software systems.

Why Software Is Inherently Complex

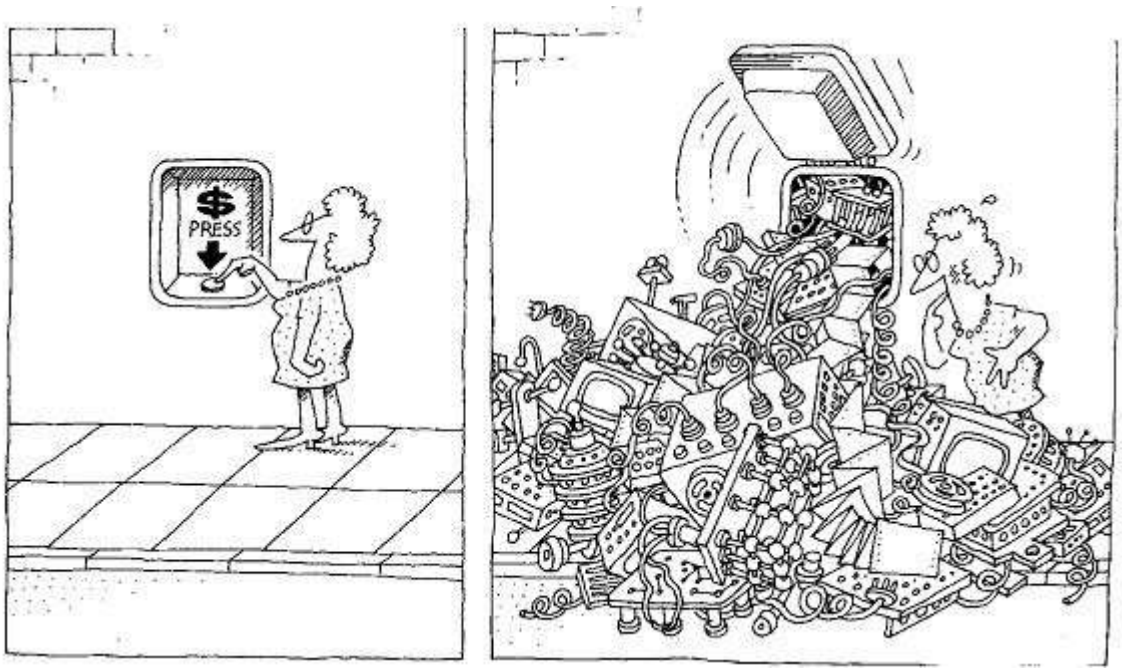
As Brooks suggests, "The complexity of software is an essential property, not an accidental one" [3]. We observe that this inherent complexity derives from four elements: the complexity of the problem domain, the difficulty of managing the developmental process, the flexibility possible through software, and the problems of characterizing the behavior of discrete systems.

The Complexity of the Problem Domain The problems we try to solve in software often involve elements of inescapable complexity, in which we find a myriad of competing,perhaps even contradictory, requirements. Consider the requirements for the electronic system of a multi-engine aircraft, a cellular phone switching system, or an autonomous robot. The raw functionality of such systems is difficult enough to comprehend, but now add all of the (often implicit) nonfunctional requirements such as usability, performance, cost, survivability, and reliability. This unrestrained external complexity is what causes the arbitrary complexity about which Brooks writes.

This external complexity usually springs from the "impedance mismatch" that exists between the users of a system and its developers: users generally find it very hard to give precise expression to their needs in a form that developers can understand In extreme cases, users may have only vague ideas of what they want in a software system. This is not so much the fault of either the users or the developers of a system; rather, it occurs because each group generally lacks expertise in the domain of the other. Users and developers have different perspectives on the nature of the problem and make different assumptions regarding the nature of the solution. Actually, even if users had perfect knowledge of their needs, we currently have few instruments for precisely capturing these requirements. The common way of expressing requirements today is with large volumes of text, occasionally accompanied by a few drawings. Such documents are difficult to comprehend, are open to varying interpretations,

and too often contain elements that are designs rather than essential requirements.

A further complication is that the requirements of a software system often change during its development, largely because the very existence of a software development project alters the rules of the problem. Seeing early products, such as design documents and prototypes, and then using a system once it is installed and operational, are forcing functions that lead users to better understand and articulate their real needs. At the same time, this process helps developers master the problem domain, enabling them to ask better questions that illuminate the dark comers of a system's desired behavior.



The task of the software development team is to engineer the illusion of simplicity.

The Difficulty of Managing the Development Process The fundamental task of the software development team is Lo engineer the illusion of simplicity - to shield users from this vast and often arbitrary external complexity. Certainly, size is no great virtue in a software system. We strive to write less code by inventing clever and powerful mechanisms that give us this illusion of simplicity, as well as by reusing frame-works of existing designs and code. However, the sheer volume of a system's requirements is sometimes inescapable and forces us cither to write a large amount of new software or to reuse existing software in novel ways. Just two decades ago, assembly language programs of only a few thousand lines of code stressed the limits of our software engineering abilities. Today, it is not unusual to find delivered systems whose size is measured in hundreds of thousands, or even millions of lines of code (and all of that in a high-order programming language, as

well). No one person can ever understand such a system completely. Even if we decompose our implementation in meaningful ways, we still end up with hundreds and sometimes thousands of separate modules. This amount of work demands that we use a team of developers, and ideally we use as small a team as possible. However, no matter what its size, there are always significant challenges associated with team development. More developers means more complex communication and hence more difficult coordination, particularly if the team is geographically dispersed, as is often the case in very large projects. With a team of developers, the key management challenge is always to maintain a unity and integrity of design.

The Flexibility Possible Through Software A home-building company generally does not operate its own tree farm from which to harvest trees for lumber; it is highly unusual for a construction firm to build an on-site steel mill to forge custom girders for a new building. Yet in the software industry such practice is common. Software offers the ultimate flexibility, so it is possible for a developer to express almost any kind of abstraction. This flexibility turns out to be an incredibly seductive property, however, because it also forces the developer to craft virtually all the primitive building blocks upon which these higher-level abstractions stand. While the construction industry has uniform building codes and standards for the quality of raw materials, few such standards exist in the software industry. As a result, software development remains a labor-intensive business.

The Problems of Characterizing the Behavior of Discrete Systems If we toss a ball into the air, we can reliably predict its path because we know that under normal conditions, certain laws of physics apply. We would be very surprised if just because we threw the ball a little harder, halfway through its flight it suddenly stopped and shot straight up into the air[2] in a not-quite-debugged software simulation of this ball's motion, exactly that kind of behavior can easily occur.

Within a large application, there may be hundreds or even thousands of variables as well as more than one thread of control. The entire collection of these variables, their current values, and the current address and calling stack of each process within the system constitute the present state of the application. Because we execute out software on digital computers, we have a system with discrete states. By contrast, analog systems such as the motion of the tossed ball are continuous systems. Parnas suggests that "when we say that a system is described by a continuous function, we are saying that it can contain no hidden surprises. Small changes in inputs will always cause correspondingly small changes in outputs" [4]. On the other hand, discrete systems by their very nature have a finite number of possible states; in large systems, there is a combinatorial explosion that makes this number very large. We try to design our systems with a separation of concerns, so that the behavior in one part of a system has minimal impact upon the behavior in another. However, the fact remains that the phase transitions among discrete states cannot be modeled by continuous functions. Each event external to a software system has the potential of placing that system in a new state, and furthermore, the mapping from state to state is not always deterministic. In the worst circumstances, an external event may corrupt the state of a system, because its designers failed to take into account certain interactions among events. For example, imagine a commercial airplane whose flight surfaces and cabin environment are managed by a single computer. We would be very unhappy if, as a result of a passenger in seat 38J turning on an overhead light, the plane immediately executed a sharp dive. In continuous systems this kind

~~of behavior would be~~ unlikely, but in discrete systems all external events can affect any part of the system's internal state. Certainly, this is the primary motivation for vigorous testing of our systems, but for all except the most trivial systems, exhaustive testing is impossible. Since we have neither the

mathematical tools nor the intellectual capacity to model the complete behavior of large discrete systems, we must be content with acceptable levels of confidence regarding their correctness.

The Consequences of Unrestrained Complexity

"The more complex the system, the more open it is to total breakdown" [5]. Rarely would a builder think about adding a new sub-basement to an existing 100-story building; to do so would be very costly and would undoubtedly invite failure. Amazingly, users of software systems rarely think twice about asking for equivalent changes. Besides, they argue, it is only a simple matter of programming.

Our failure to master the complexity of software results in projects that are late, over budget, and deficient in their stated requirements. We often call this condition the *software crisis*, but frankly, a malady that has carried on this long must be called normal. Sadly, this crisis translates into the squandering of human resources - a most precious commodity - as well as a considerable loss of opportunities. There are simply not enough good developers around to create all the new software that users need. Furthermore, a significant number of the developmental personnel in any given organization must often be dedicated to the maintenance or preservation of geriatric software. Given the indirect as well as the direct contribution of software to the economic base of most industrialized countries, and considering the ways in which software can amplify the powers of the individual, it is unacceptable to allow this situation to continue.

How can we change this dismal picture? Since the underlying problem springs from the inherent complexity of software, our suggestion is to first study how complex systems in other disciplines are organized. Indeed, if we open our eyes to the world about us, we will observe successful systems of significant complexity. Some of these systems are the works of humanity, such as the Space Shuttle, the England/France tunnel, and large business organizations such as Microsoft or General Electric. Many even more complex systems appear in nature, such as the human circulatory system or the structure of a plant.

The Structure of Complex Systems

Examples of Complex Systems

The Structure of a Personal Computer A personal computer is a device of moderate complexity. Most of them are composed of the same major elements: a central processing unit (CPU), a monitor, a keyboard, and some sort of secondary storage device, usually either a floppy disk or a hard disk drive. We may take any one of these parts and further decompose

it. For example, a CPU typically encompasses primary memory, an arithmetic/logic unit (ALU), and a bus to which peripheral devices are attached. Each of these parts may in turn be further decomposed: an ALU may be divided into registers and random control logic, which themselves are constructed from even more primitive elements, such as NAND gates, inverters, and so on.

Here we see the hierarchic nature of a complex system. A personal computer functions properly only because of the collaborative activity of each of its major parts. Together, these separate parts logically form a whole. Indeed, we can reason about how a computer works only because we can decompose it into parts that we can study separately. Thus, we may study the operation of a monitor independently of the operation of the hard disk drive. Similarly, we may study the ALU without regard for the primary memory subsystem.

Not only are complex systems hierarchic, but the levels of this hierarchy represent different levels of abstraction, each built upon the other, and each understandable by itself. At each level of abstraction, we find a collection of devices that collaborate to provide services to higher layers. We choose a given level of abstraction to suit our particular needs. For instance, if we were trying to track down a timing problem in the primary memory, we might properly look at the gate-level architecture of the computer, but this level of abstraction would be inappropriate if we were trying to find the source of a problem in a spreadsheet application.

The Structure of Plants and Animals In botany, scientists seek to understand the similarities and differences among plants through a study of their morphology, that is, their form and structure. Plants are complex multicellular organisms, and from the cooperative activity of various plant organ systems arise such complex behaviors as photosynthesis and transpiration.

Plants consist of three major structures (roots, stems, and leaves), and each of these has its own structure. For example, roots encompass branch roots, root hairs, the root apex, and the root cap. Similarly, a cross-section of a leaf reveals its epidermis, mesophyll, and vascular tissue. Each of these structures is further composed of a collection of cells, and inside each cell we find yet another level of complexity, encompassing such elements as chloroplasts, a nucleus, and so on. As with the structure of a computer, the parts of a plant form a hierarchy, and each level of this hierarchy embodies its own complexity.

All parts at the same level of abstraction interact in well-defined ways. For example, at the highest level of abstraction, roots are responsible for absorbing water and minerals from the soil. Roots interact with stems, which transport these raw materials up to the leaves. The leaves in turn use the water and minerals provided by the stems to produce food through photosynthesis.

There are always clear boundaries between the outside and the inside of a given level. For example, we can state that the parts of a leaf work together to provide

the functionality of the leaf as a whole, and yet have little or no direct interaction with the elementary parts of the

roots. In simpler terms, there is a clear separation of concerns among the parts at different levels of abstraction.

In a computer, we find NAND gates used in the design of the CPU as well as in the hard disk drive. Likewise, a considerable amount of commonality cuts across all parts of the structural hierarchy of a plant. This is God's way of achieving an economy of expression. For example, cells serve as the basic building blocks in all structures of a plant; ultimately, the roots, stems, and leaves of a plant are all composed of cells. Yet, although each of these primitive elements is indeed a cell, there are many different kinds of cells. For example, there are cells with and without chloroplasts, cells with walls that are impervious to water and cells with walls that are permeable, and even living cells and dead cells.

In studying the morphology of a plant, we do not find individual parts that are each responsible for only one small step in a single larger process, such as photosynthesis. In fact, there are no centralized parts that directly coordinate the activities of lower level ones. Instead, we find separate parts that act as independent agents, each of which exhibits some fairly complex behavior, and each of which contributes to many higher-level functions. Only through the mutual cooperation of meaningful collections of these agents do we see the higher-level functionality of a plant. The science of complexity calls this *emergent behavior*: The behavior of the whole is greater than the sum of its parts [6].

Turning briefly to the field of zoology, we note that multicellular animals exhibit a hierarchical structure similar to that of plants: collections of cells form tissues, tissues work together as organs, clusters of organs define systems (such as the digestive system), and so on. We cannot help but again notice God's awesome economy of expression: the fundamental building block of all animal matter is the cell, just as the cell is the elementary structure of all plant life. Granted, there are differences between these two. For example, plant cells are enclosed by rigid cellulose walls, but animal cells are not. Notwithstanding these differences, however, both of these structures are undeniably cells. This is an example of commonality that crosses domains.

A number of mechanisms above the cellular level are also shared by plant and animal fife. For example, both use some sort of vascular system to transport nutrients within the organism, and both exhibit differentiation by sex among members of the same species.

The Structure of Matter The study of fields as diverse as astronomy and nuclear physics provides us with many other examples of incredibly complex systems. Spanning these two disciplines, we find yet another structural hierarchy. Astronomers study galaxies that are arranged in clusters, and stars, planets, and various debris are the constituents of galaxies. Likewise, nuclear physicists are concerned with a structural hierarchy, but one on an entirely different scale. Atoms are made up of electrons, protons, and neutrons; electrons appear to be

elementary particles, but protons, neutrons, and other particles are formed from more basic components called *quarks*.

Again we find that a great commonality in the form of shared mechanisms unifies this vast hierarchy. Specifically, there appear to be only four distinct kinds of forces at work in the universe: gravity, electromagnetic interaction, the strong force, and the weak force. Many laws of physics involving these elementary forces, such as the laws of conservation of energy and of momentum, apply to galaxies as well as quarks.

The Structure of Social Institutions As a final example of complex systems, we turn to the structure of social institutions. Groups of people join together to accomplish tasks that cannot be done by individuals. Some organizations are transitory, and some endure beyond many lifetimes. As organizations grow larger, we see a distinct hierarchy emerge. Multinational corporations contain companies, which in turn are made up of divisions, which in turn contain branches, which in turn encompass local offices, and so on. If the organization endures, the boundaries among these parts may change, and over time, a new, more stable hierarchy may emerge.

The relationships among the various parts of a large organization are just like those found among the components of a computer, or a plant, or even a galaxy. Specifically, the degree of interaction among employees within an individual office is greater than that between employees of different offices. A mail clerk usually does not interact with the chief executive officer of a company but does interact frequently with other people in the mail room. Here too, these different levels are unified by common mechanisms. The clerk and the executive are both paid by the same financial organization, and both share common facilities, such as the company's telephone system, to accomplish their tasks.

The Five Attributes of a Complex System

Drawing from this line of study, we conclude that there are five attributes common to all complex systems. Building upon the work of Simon and Ando, Courtois suggests the following:

> *"Frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of elementary components is reached" [7].*

Simon points out that "the fact that many complex systems have a nearly decomposable, hierarchic structure is a major facilitating factor enabling us to understand, describe, and even 'see' such systems and their parts" [8]. Indeed, it is likely that we can understand only those systems that have a hierarchic structure.

It is important to realize that the architecture of a complex system is a function of its components as well as the hierarchic relationships among these components. As Rechtin observes, "All systems have subsystems and all systems are parts of larger systems . . . The valued added by a system must come from the relationships between the parts, not from the parts per se" [9].

Regarding the nature of the primitive components of a complex system, our experience suggests that

> *The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system.*

What is primitive for one observer may be at a much higher level of abstraction for another.

Simon calls hierarchic systems decomposable, because they can be divided into identifiable parts; he calls them nearly decomposable, because their parts are not completely independent. This leads us to another attribute common to all complex systems:

> *"Intracomponent linkages are generally stronger than intercommoning linkages. This fact has the effect of separating the high-frequency dynamics of the components - involving the internal structure of the components - from the low-frequency dynamics - involving interaction among components"[10].*

This difference between intra- and intercomponent interactions provides a clear separation of concerns among the various parts of a system, making it possible to study each part in relative isolation.

As we have discussed, many complex systems are implemented with an economy of expression. Simon thus notes that

> *"Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements " [11].*

In other words, complex systems have common patterns. These patterns may involve the reuse of small components, such as the cells found in both plants and animals, or of larger structures, such as vascular systems, also found in both plants and animals.

Earlier, we noted that complex systems tend to evolve over time. As Simon suggests, "complex systems will evolve from simple systems much more rapidly if there are stable intermediate forms than if there are not" [12]. In more dramatic terms, Gall states that

> *"A complex system that works is invariably found to have evolved from a simple system that worked.... A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system " [13].*

As systems evolve, objects that were once considered complex become the primitive objects upon which more complex systems are built. Furthermore, we

can never craft these primitive objects correctly the first time. We must use them in context first, and then improve them over time as we learn more about the real behavior of the system.

Organized and Disorganized Complexity

The Canonical Form of a Complex System The discovery of common abstractions and mechanisms greatly facilitates our understanding of complex systems. For example, with just a few minutes of orientation, an experienced pilot can step into a multiengine jet aircraft he or she has never flown before and safely fly the vehicle. Having recognized the properties common to all such aircraft, such as the functioning of the rudder, ailerons, and throttle, the pilot primarily needs to learn what properties are unique to that particular aircraft. If the pilot already knows how to fly a given aircraft, it is far easier to know how to fly a similar one.

This example suggests; that we have been using the term *hierarchy* in a rather loose fashion. Most interesting systems do not embody a single hierarchy; instead, we find that many different hierarchies are usually present within the same complex system. For example, an aircraft may be studied by decomposing it into its propulsion system, flight-control system, and so on. This decomposition represents a structural, or "part of" hierarchy. Alternately, we can cut across the system in an entirely orthogonal way. For example, a turbofan engine is a specific kind of jet engine, and a Pratt and Whitney TF30 is a specific kind of turbofan engine. Stated another way, a jet engine represents a generalization of the properties common to every kind of jet engine; a turbofan engine is simply a specialized kind of jet engine, with properties that distinguish it, for example, from ramjet engines.

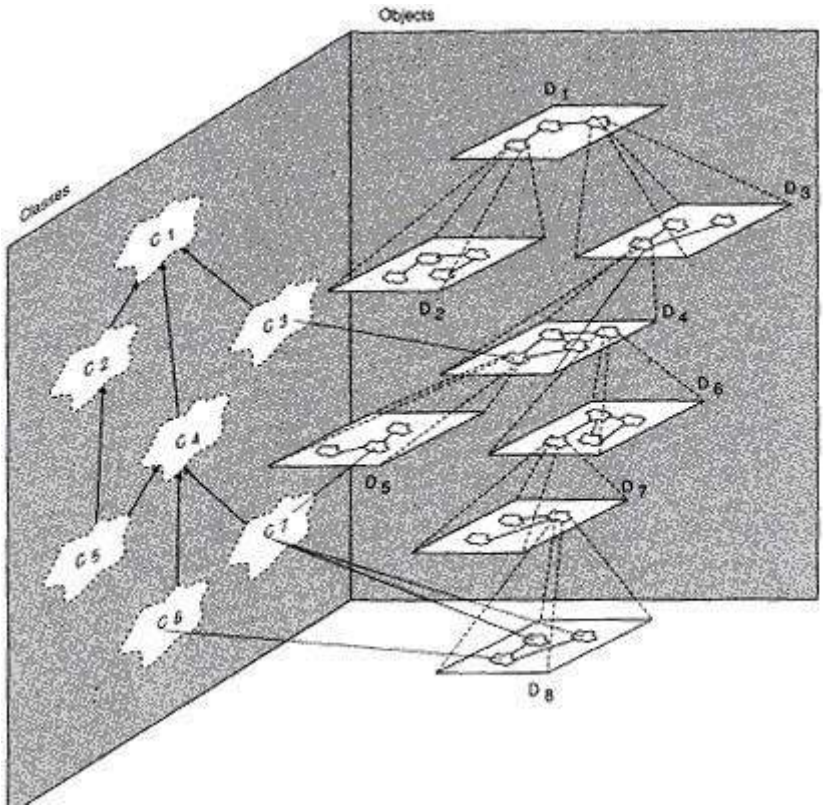


Figure 1-1
The Canonical Form of a Complex System

This second hierarchy represents an "is a" hierarchy. In our experience, we have found it essential to view a system from both perspectives, studying its "is a" hierarchy as well as its "part of" hierarchy. For reasons that will become clear in the next chapter, we call these hierarchies the *class structure* and the *object structure*, respectively[3].

Combining the concept of the class and object structure together with the five attributes of a complex system, we find that virtually all complex systems take en the same (canonical) form, as we show in Figure 1-1. Here we see the two orthogonal hierarchies of the system: its class structure and its object structure. Each hierarchy is layered, with the more abstract classes and objects built upon more primitive ones. What class or object is chosen as primitive is relative to the problem at hand, Especially among the parts of the object structure, there are close collaborations among objects at the same level of abstraction, Looking inside any given level reveals yet another level of complexity. Notice also that the class structure and the object structure are not completely independent; rather, each object in the object structure represents a specific instance of some class. As the figure suggests, there are usually many more objects than classes of objects within a complex system. Thus, by showing the "part of" as well as the "is a" hierarchy, we explicitly expose the redundancy of the system under consideration, lf we did not reveal a system's class structure, we would have to duplicate our knowledge about the properties of each individual part. With the inclusion of the class structure, we capture these common properties in one place.

Our experience is that the most successful complex software systems are those whose designs explicitly encompass a well-engineered class and object structure and whose structure embodies the five attributes of complex systems described in the previous section. Lest the importance of this observation be missed, let us be even more direct: we very rarely encounter software systems that are delivered on time, within budget, and that meet their requirements, unless they are designed with these factors in mind.

Collectively, we speak of the class and object structure of a system as its *architecture*.

The Limitations of the Human Capacity for Dealing with Complexity If we know what the design of complex software systems should be like, then why do we still have serious problems in successfully developing them? As we discuss in the next chapter, this concept of the organized complexity of software (whose guiding principles we call the *object model*) is relatively new. However, there is yet another factor that dominates: the fundamental limitations of the human capacity for dealing with complexity.

As we first begin to analyze a complex software system, we find many parts that must interact in a multitude of intricate ways, with little perceptible commonality among either the parts or their interactions: this is an example of disorganized complexity. As we work to bring organization to this complexity through the process of design, we must think about many things at once. For example, in an air traffic control system, we must deal with the state

of many different aircraft at once, involving such properties as their location, speed, and heading. Especially in the case of discrete systems, we must cope with a fairly large, intricate, and sometimes no deterministic state space. Unfortunately, it: is absolutely impossible for a single person to keep track of all of these details at once. Experiments by psychologists, such as those of Miller, suggest that the maximum number of chunks of information that an individual can simultaneously comprehend is on the order of seven, plus or minus two [14]. This channel capacity seems to be related to the capacity of short-term

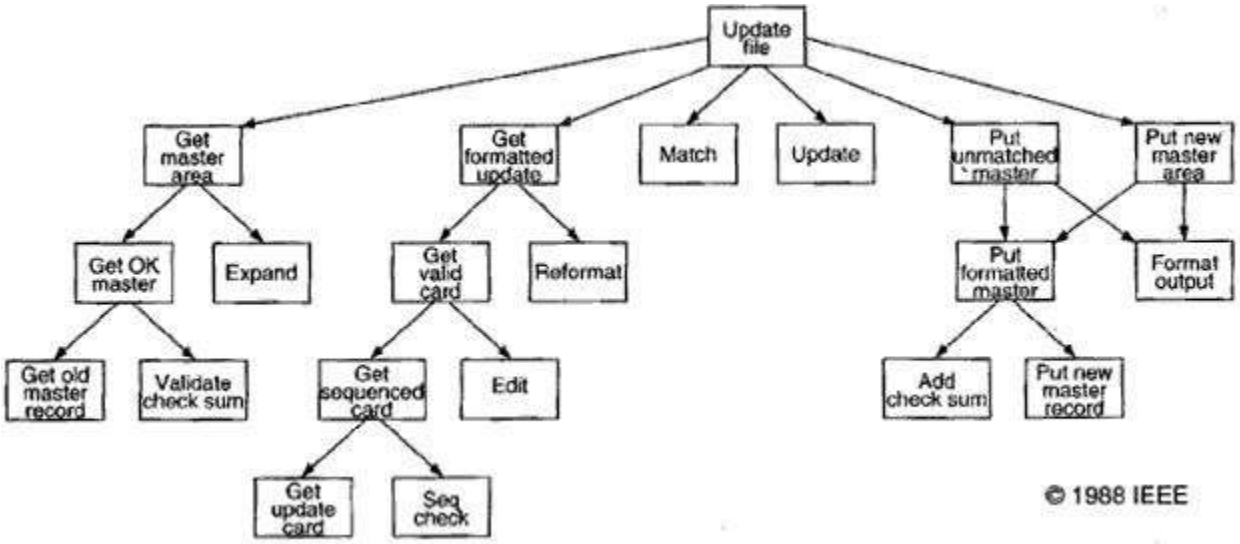


Figure 1-2
Algorithmic Decomposition

memory. Simon additionally notes that processing speed is a limiting factor: it takes the mind about five seconds to accept a new chunk of information [15]

We are thus faced with a fundamental dilemma. The complexity of the software systems we are asked to develop is increasing, yet there are basic limits upon our ability to cope with this complexity. How then do we resolve this predicament?

Bringing Order to Chaos

The Role of Decomposition

As Dijkstra suggests, “The technique of mastering complexity has been known since ancient times: *divide et impera* (divide and rule)" . When designing a complex software system, it is essential to decompose it into smaller and smaller parts, each of which we may then refine independently. In this manner, we satisfy the very real constraint that exists upon the channel capacity of human cognition: to understand any given level of a system, we need only comprehend a few parts (rather than all parts) at once. Indeed, as Parnas observes, intelligent

decomposition directly addresses the inherent complexity of software by forcing a division of a system's state space .

Algorithmic Decomposition Most of us have been formally trained in the dogma of top-down structured design, and so we approach decomposition as a simple matter of algorithmic decomposition, wherein each module in the system denotes a major step in some overall process. Figure 1-2 is an example of one of the products of structured design, a structure chart that shows the relationships among various functional elements of the solution. This particular structure chart illustrates part of the design of a program that updates the

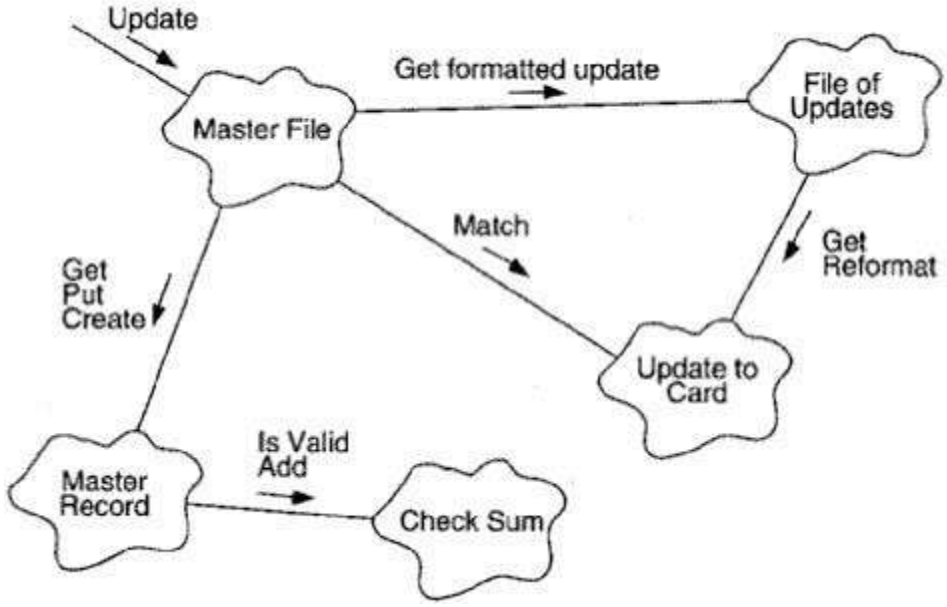


Figure 1-3
Object-Oriented Decomposition

content of a master file. It was automatically generated from a data flow diagram by an expert system tool that embodies the rules of structured design [18].

Object-Oriented Decomposition We suggest that there is an alternate decomposition possible for the same problem. In Figure 1-3, we have decomposed the system according to the key abstractions in the problem domain. Rather than decomposing the problem into steps such as *Get formatted update* and *Add check sum* , we have identified objects such as *Master File* and *Check Sum*, which derive directly from the vocabulary of the problem domain.

Although both designs solve the same problem, they do so in quite different ways. In this second decomposition, we view the world as a set of autonomous agents that collaborate to perform some higher level behavior. *Get formatted update* thus does not exist as an independent algorithm; rather, it is an operation associated with the object *File of Updates*. Calling this operation creates another object, *Update to Card*. In this manner, each object in our solution embodies its own unique behavior, and each one models some object in the real world. From this perspective, an object is simply a tangible entity which exhibits some well-defined behavior. Objects do things, and we ask them to perform what they do by sending

them messages. Because our decomposition is based upon objects and not algorithms, we call this an *object-oriented* decomposition.

Algorithmic versus Object-Oriented Decomposition Which is the right way to decompose a complex system - by algorithms or by objects? Actually, this is a trick question, because the right answer is that both views are important: the algorithmic view highlights the ordering of events, and the object-oriented view emphasizes the agents that either cause action or are the subjects upon which these operations act. However, the fact remains that we cannot construct a complex system in both ways simultaneously, for they are completely orthogonal views[4]. We must start decomposing a system either by algorithms or by objects, and then use the resulting structure as the framework for expressing the other perspective.

Our experience leads us to apply the object-oriented view first because this approach is better at helping us organize the inherent complexity of software systems, just as it helped us to describe the organized complexity of complex systems as diverse as computers, plants, galaxies, and large social institutions. As we will discuss further in Chapters 2 and 7, object-oriented decomposition has a number of highly significant advantages over algorithmic decomposition. Object-oriented decomposition yields smaller systems through the reuse of common mechanisms, thus providing an important economy of expression. Object-oriented systems are also more resilient to change and thus better able to evolve over time, because their design is based upon stable intermediate forms. Indeed, object-oriented decomposition greatly reduces the risk of building complex software systems, because they are designed to evolve incrementally from smaller systems in which we already have confidence. Furthermore, object-oriented decomposition directly addresses the inherent complexity of software by helping us make intelligent decisions regarding the separation of concerns in a large state space.

The Role of Abstraction

Earlier, we referred to Miller's experiments, from which he concluded that an individual can comprehend only about seven, plus or minus two, chunks of information at one time. This number appears to be independent of information content. As Miller himself observes, "The span of absolute judgment and the span of immediate memory impose severe limitations on the amount of information that we are able to receive, process and remember. By organizing the stimulus input simultaneously into several dimensions and successively into a sequence of chunks, we manage to break ... this informational bottleneck" [35]. In contemporary terms, we call this process chunking, or *abstraction*.

As Wulf describes it, "We (humans) have developed an exceptionally powerful technique for dealing with complexity. We abstract from it. Unable to master the entirety of a complex object, we choose to ignore its inessential details, dealing instead with the generalized, idealized model of the object [36]. For example, when studying how photosynthesis works in

a plant, we can focus upon the chemical reactions in certain cells in a leaf, and ignore all other parts, such as the roots and stems. We are still constrained by the number of things that we can comprehend at one time, but through abstraction, we use chunks of information with increasingly greater semantic content. This is especially true if we take an object-oriented view of the world, because objects, as abstractions of entities in the real world, represent a particularly dense and cohesive clustering of information. Chapter 2 examines the meaning of abstraction in much greater detail.

The Role of Hierarchy

Another way to increase the semantic content of individual chunks of information is by explicitly recognizing the class and object hierarchies within a complex software system. The object structure is important because it illustrates how different objects collaborate with one another through patterns of interaction that we call *mechanisms*. The class structure is equally important, because it highlights common structure and behavior within a system. Thus, rather than study each individual photosynthesizing cell within a specific plant leaf, it is enough to study one such cell, because we expect that all others will exhibit similar behavior. Although we treat each instance of a particular kind of object as distinct, we may assume that it shares the same behavior as all other instances of that same kind of object. By classifying objects into groups of related abstractions (for example, kinds of plant cells versus animal cells), we come to explicitly distinguish the common and distinct properties of different objects, which further helps us to master their inherent complexity .

Identifying the hierarchies within a complex software system is often not easy, because it requires the discovery of patterns among many objects, each of which may embody some tremendously complicated behavior. Once we have exposed these hierarchies, however, the structure of a complex system, and in turn our understanding of it, becomes vastly simplified. Chapter 3 considers in detail the nature of class and object hierarchies, and Chapter 4 describes techniques that facilitate our identification of these patterns.

On Designing Complex Systems

Engineering as a Science and an Art

The practice of every engineering discipline - be it civil, mechanical, chemical, electrical, or software engineering - involves elements of both science and art. As Petroski eloquently states, "The conception of a design for a new structure can involve as much a leap of the imagination and as much a synthesis of experience and knowledge as any artist is required to bring to his canvas or paper. And once that design is articulated by the engineer as artist, it must be analyzed by the engineer as scientist in as rigorous an application of the scientific method as any scientist must make". Similarly, Dijkstra observes that "the programming challenge is a large-scale exercise in applied abstraction and thus requires the

abilities of the formal mathematician blended with the attitude of the competent engineer."

The role of the engineer as artist is particularly challenging when the task is to design an entirely new system. Frankly, this is the most common circumstance in software engineering. Especially in the case of reactive systems and systems for command and control, we are frequently asked to write software for an entirely unique set of requirements, often to be executed on a configuration of target processors constructed specifically for this system. In other cases, such as the creation of frameworks, tools for research in artificial intelligence, or even information management systems, we may have a well defined, stable target environment, but our requirements may stress the software technology in one or more dimensions. For example, we may be asked to craft systems that are faster, have greater capacity, or have radically improved functionality. In all these situations, we try to use proven abstractions and mechanisms (the "stable intermediate forms," in Simon's words) as a foundation upon which to build new complex systems. In the presence of a large library of reusable software components, the software engineer must assemble these parts in innovative ways to satisfy the stated and implicit requirements, just as the painter or the musician must push the limits of his or her medium. Unfortunately, since such rich libraries rarely exist for the software engineer, he or she must usually proceed with a relatively primitive set of facilities.

The Meaning of Design

In every engineering discipline, design encompasses the disciplined approach we use to invent a solution for some problem, thus providing a path from requirements to implementation. In the context of software engineering, Mostow suggests that the purpose of design is to construct a system that:

> "Satisfies a given (perhaps informal) functional specification
> Conforms to limitations of the target medium
> Meets implicit or explicit requirements on performance and resource usage
> Satisfies implicit or explicit design criteria on the form of the artifact
> Satisfies restrictions on the design process itself, such as its length or cost, or the tools available for doing the design"

As Stroustrup suggests, "the purpose of design is to create a clean and relatively simple internal structure, sometimes also called an architecture.... A design is the end product of the
design process" . Design involves balancing a set of competing requirements. The products of design are models that enable us to reason about our structures, make trade-offs when requirements conflict, and in general, provide a blueprint for implementation.

The Importance of Model Building The building of models has a broad acceptance among all engineering disciplines, largely because model building appeals to the principles of decomposition, abstraction, and hierarchy . Each model within a design describes a specific aspect of the system under consideration. As much as

possible, we seek to build new models upon old models in which we already have confidence. Models give us the

opportunity to fail under controlled conditions. We evaluate each model under both expected and unusual situations, and then alter them when they fail to behave as we expect or desire.

We have found that in order to express all the subtleties of a complex system, we must use more than one kind of model. For example, when designing a single-board computer, an electrical engineer must take into consideration the gate-level view of the system as well as the physical layout of integrated circuits on the board. This gate-level view forms a logical picture of the design of the system, which helps the engineer to reason about the cooperative behavior of the gates. The board layout represents the physical packaging of these gates, constrained by the board size, available power, and the kinds of integrated circuits that exist. From this view, the engineer can independently reason about factors such as heat dissipation and manufacturability. The board designer must also consider dynamic as well as static aspects of the system under construction. Thus, the electrical engineer uses diagrams showing the static connections among individual gates, as well as timing diagrams that show the behavior of these gates over time. The engineer can then employ tools such as oscilloscopes and digital analyzers to validate the correctness of both the static and dynamic models.

The Elements of Software Design Methods Clearly, there is no magic, no "silver bullet" [43], that: can unfailingly lead the software engineer down the path from requirements to the implementation of a complex software system. In fact, the design of complex software systems does not lend itself at all to cookbook approaches. Rather, as noted earlier in the fifth attribute of complex systems, the design of such systems involves an incremental and iterative process.

Still, sound design methods do bring some much-needed discipline to the development process. The software engineering community has evolved dozens of, different design methods, which we can loosely classify into three categories (see sidebar). Despite their differences, all of these methods have elements in common. Specifically, each method includes the following:

- Notation — The language for expressing each model
- Process — The activities leading to the orderly construction of the system's models
- Tools — The artifacts that eliminate the tedium of model building and enforce rules about the models themselves, so that errors and inconsistencies can be exposed

A sound design method is based upon a solid theoretical foundation, yet offers degrees of freedom for artistic innovation.

The Models of Object-Oriented Development Is there a "best" design method? No, there is no absolute answer to this question, which is actually just a veiled way of

asking the earlier question: What is the best way to decompose a complex system? To reiterate, we have found great value in building models

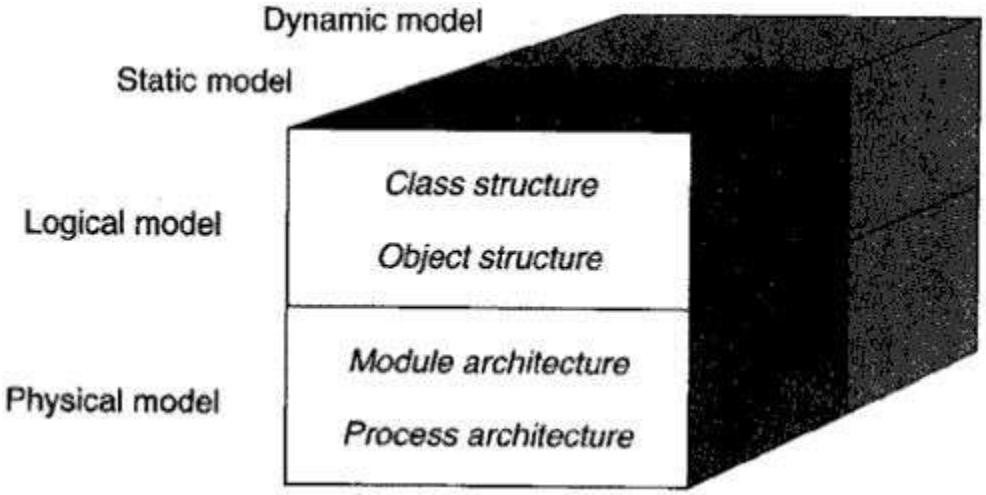


Figure 1-4
The Models of Object-Oriented Development

that are focused upon the "things" we find, in the problem space, forming what we refer to as an *object-oriented decomposition*.

Object-oriented analysis and design is the method that leads us to an object-oriented decomposition. By applying object-oriented design, we create software that is resilient to change and written with economy of expression. We achieve a greater level of confidence in the correctness of our software through an intelligent separation of its state space. Ultimately, we reduce the risks that are inherent in developing complex software systems.

Because model building is so important to the systems, object-oriented development offers a rich describe in Figure 1-4. The models of object-oriented analysis and design reflect the importance of explicitly capturing both the class and object hierarchies of the system under design. These models also cover the spectrum of the important design decisions that we must consider in developing a complex system, and so encourage us to craft implementations that embody the five attributes of well-formed complex systems.

The Object Model

The Evolution of the Object Model

Trends in Software Engineering

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

Most new industrial-strength software systems are larger and more complex than their predecessors were even just a few years ago. This growth in complexity has prompted a significant amount of useful applied research in software engineering, particularly with regard to decomposition, abstraction, and hierarchy. The development of more expressive programming languages has complemented these advances. The trend has been a move away from languages that tell the computer what to do (imperative languages) toward languages that describe the key abstractions in the problem domain (declarative languages).

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

First-Generation Languages (1954-1958)

FORTRANI	Mathematical expressions
ALGOL 58	Mathematical expressions
Flowmatic	Mathematical expressions
IPL V	Mathematical expressions

Second-Generation Languages (1959~1961)

FORTRANII	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)

Simula
PL/1
ALGOL 68
Pascal

	Rigorous successor to ALGOL 60
FORTRAN + ALGOL + COBOL	Simple successor to ALGOL 60 Classes, data abstraction

The Generation Gap (1970-1980)

Many different languages were invented, but few endured

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 1 were thus developed to allow the programmer to write mathematical formulas, thereby freeing the programmer from some of the intricacies of assembly or machine language. This first generation of high-order programming languages therefore represented a step closer to the problem space, and a step further away from the

underlying machine. Among second-generation languages, the emphasis was upon algorithmic abstractions. By this time, machines were becoming more and more powerful, and the economics of the computer industry meant that more kinds of problems could be automated, especially for business applications. Now, the focus was largely upon telling the machine what to do: read these personnel records first, sort them next, and then print this report. Again, this new generation of high-order programming languages moved us a step closer to the problem space, and further away from the underlying machine. By the late 1960s, especially with the advent of transistors and then integrated circuit technology, the cost of computer hardware had dropped dramatically, yet processing capacity had grown almost exponentially. Larger problems could now be solved, but these demanded the manipulation of more kinds of data, Thus, languages such as ALGOL 60 and, later, Pascal evolved with support for data abstraction. Now a programmer could describe the meaning of related kinds of data (their type) and let the programming language enforce these design decisions. This generation of high-order programming languages again moved our software a step closer to the problem domain, and further away from the underlying machine.

The 1970s provided us with a frenzy of activity in programming language research, resulting in the creation of literally a couple of thousand different programming languages and their dialects. To a large extent, the drive to write larger and larger programs highlighted the inadequacies of earlier languages; thus, many new language mechanisms were developed to address these limitations. Few of these languages survived (have you seen a recent textbook on the languages Fred, Chaos, or Tranquil?); however, many of the concepts that they introduced found their way into successors of earlier languages. Thus, today we have Smalltalk (a revolutionary successor to Simula), Ada (a successor to ALGOL 68 and Pascal, with contributions from Simula, Alphard, and CLU),

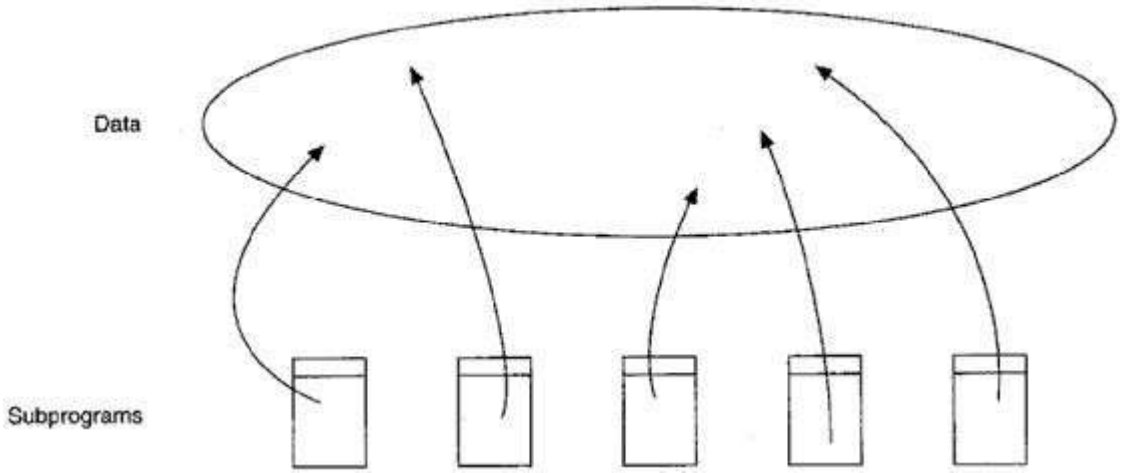


Figure 2-1
The Topology of First- and Early Second-Generation Programming Languages

CLOS (which evolved from Lisp, LOOPS, and Flavors), C++ (derived from a marriage of C and Simula), and Eiffel (derived from Simula and Ada). What is of the greatest interest to us is the class of languages we call object-based and object-oriented. *Object-based* and *object-oriented* programming languages best support the object-oriented decomposition of software.

The Topology of First- and Early Second-Generation Programming Languages To show precisely what we mean, let's study the structure of each generation of programming languages. In Figure 2-1, we see the topology of most first-and early second-generation programming languages. By topology, we mean the basic physical building blocks of the language and how those parts can be connected. In this figure, we see that for languages such as FORTRAN and COBOL, the basic physical building block of all applications is the subprogram (or the paragraph, for those who speak COBOL). Applications written in these languages exhibit a relatively flat physical structure, consisting only of global data and subprograms. The arrows in this figure indicate dependencies of the subprograms on various data. During design, one can logically separate different kinds of data from one another, but there is little in these languages that can enforce these design decisions. An error in one part of a program can have a devastating ripple effect across the rest of the system, because the global data structures are exposed for all subprograms to see. 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.

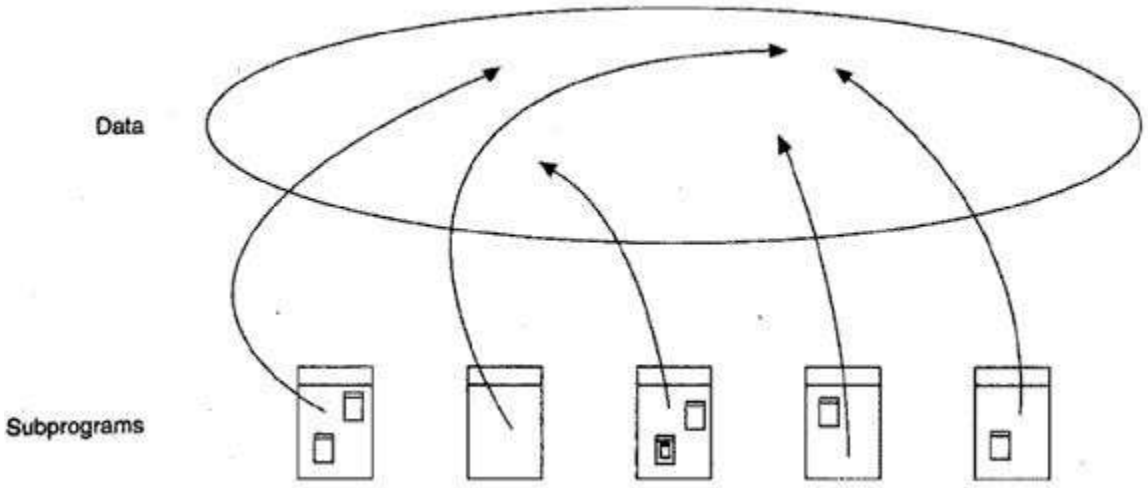


Figure 2-2
The Topology of Late Second- and Early Third-Generation Programming Languages

The Topology of Late Second- and Early Third-Generation Programming Languages By the mid-1960s, programs were finally being recognized as important intermediate points between the problem and the computer [3]. As Shaw points out, "The first software abstraction, now called the 'procedural' abstraction, grew directly out of this pragmatic view of software. . . . Subprograms were invented prior to 1950, but were not fully appreciated as abstractions at the time. . . . Instead, they were originally seen as labor-saving devices.... Very
quickly though, subprograms were appreciated as a way to abstract program functions" . The realization that subprograms could serve as an abstraction mechanism had three important consequences. First, languages were invented that supported a variety of parameter passing mechanisms. Second, the foundations of structured programming were

laid, manifesting themselves in language support for the nesting of subprograms and the development of theories regarding control structures and the scope and visibility of declarations. Third, structured design methods emerged, offering guidance to designers trying to build large systems using subprograms as basic physical building blocks. Thus, it is not surprising, as Figure 2-2 shows, that the topology of late second- and early third-generation languages is largely a variation on the theme of earlier generations. This topology addresses some of the inadequacies of earlier languages, namely, the need to have greater control over algorithmic abstractions, but it still fails to address the problems of programming-in-the-large and data design.

The Topology of Late Third-Generation Programming Languages Starting with FORTRAN II, and appearing in most late third-generation program languages, another important structuring mechanism evolved to address the growing issues of programming-in-the-large. Larger programming projects meant larger

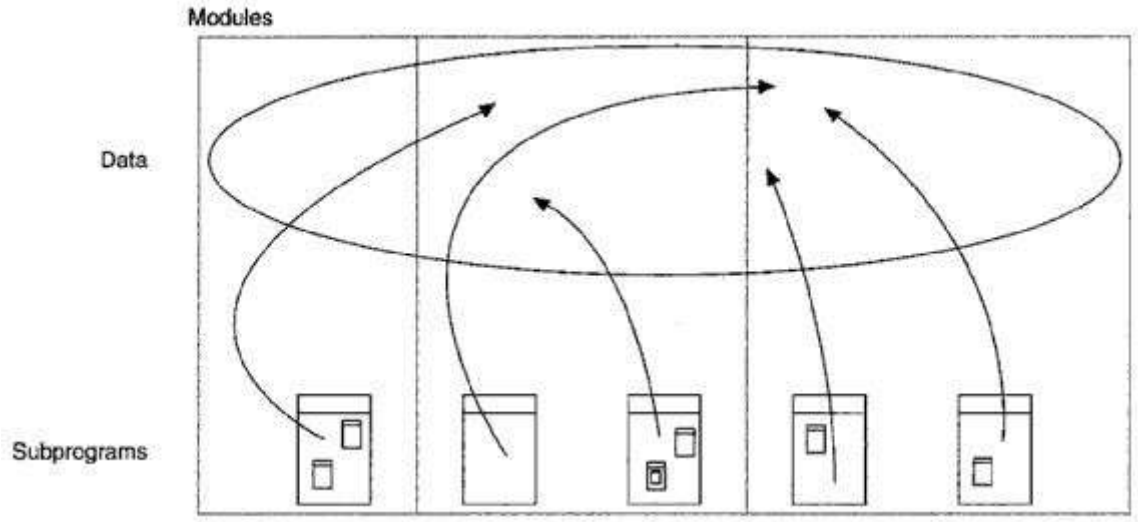


Figure 2-3
The Topology of Late Third-Generation Programming Languages

development teams, and thus the need to develop different parts of the same program independently. The answer to this need was the separately compiled module, which in its early conception was little more than an arbitrary container for data and subprograms, as Figure 2-3 shows. Modules were rarely recognized as an important abstraction mechanism; in practice they were used simply to group subprograms that were most likely to change together. Most languages of this generation, while supporting some sort of modular structure, had few rules that required semantic consistency among module interfaces. A developer writing a subprogram for one module might assume that it would be called with three different parameters: a floating-point number, an array of ten elements, and an integer representing a Boolean flag. In another module, a call to this subprogram might incorrectly use actual parameters that: violated these assumptions: an integer, an array of five elements, and a negative number. Similarly, one module might use a block of common data which 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 The importance of data abstraction to mastering complexity is clearly stated by Shankar: "The nature of abstractions that may be achieved through the use of procedures is well suited to the description of abstract operations, but is not particularly well suited to the description of abstract objects. This is a serious drawback, for in many applications, the complexity of the data objects to be manipulated contributes substantially to the overall complexity of the problem,' . This realization had two important consequences. First, data-driven design

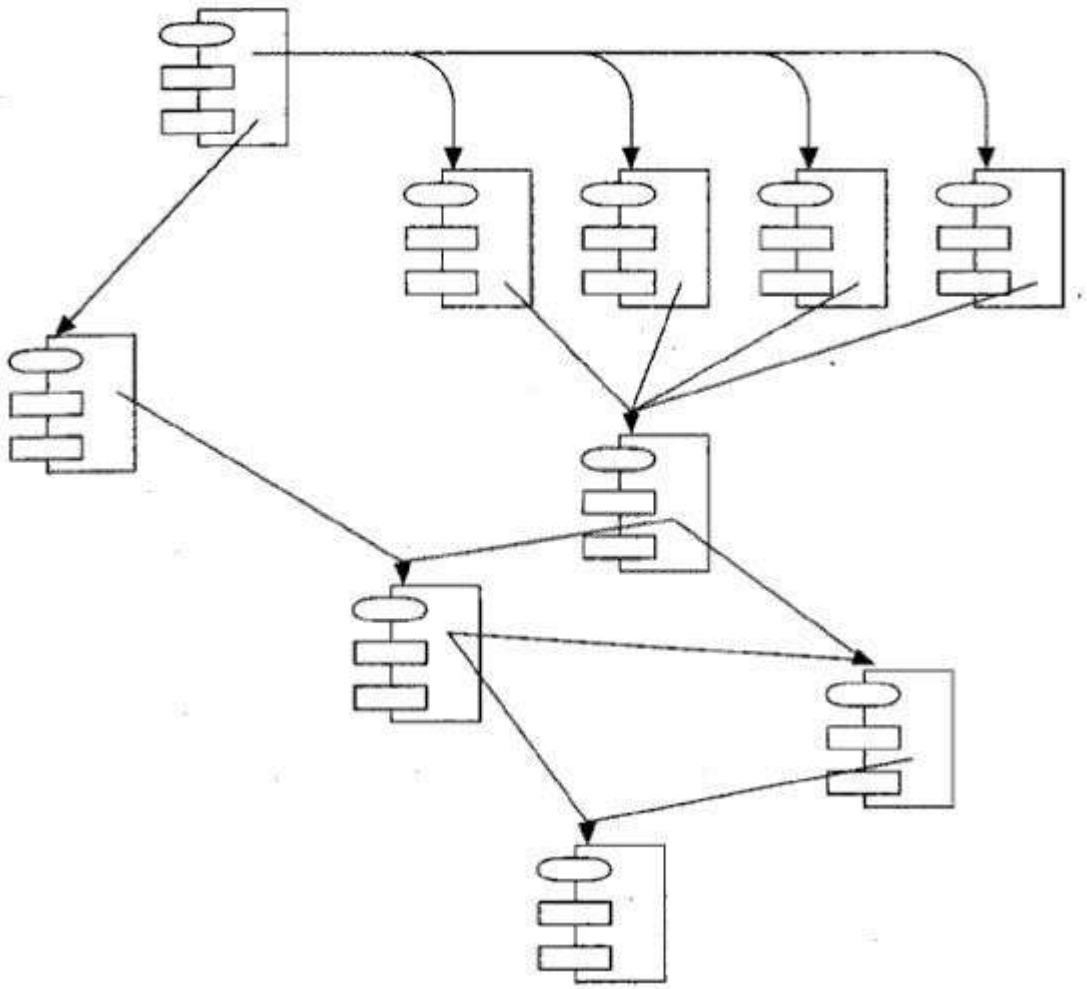


Figure 2-4
The Topology of Small- to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

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 during the period of the language generation gap, resulting in the relatively recent development of several languages such as -Smalltalk, Object Pascal, C++, CLOS, Ada, and Eiffel. For reasons that we will explain shortly, these languages are called *object-based or object-oriented*. Figure 2-4 illustrates the topology of these languages for small- to moderate-sized applications. The physical building block in these languages is the *module*, which represents a logical collection of classes and objects instead of subprograms, as in earlier languages. To state it another way, "If procedures and functions are verbs and pieces of data are nouns, a procedure-oriented program is organized around verbs while an object-oriented program is organized around nouns" . 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

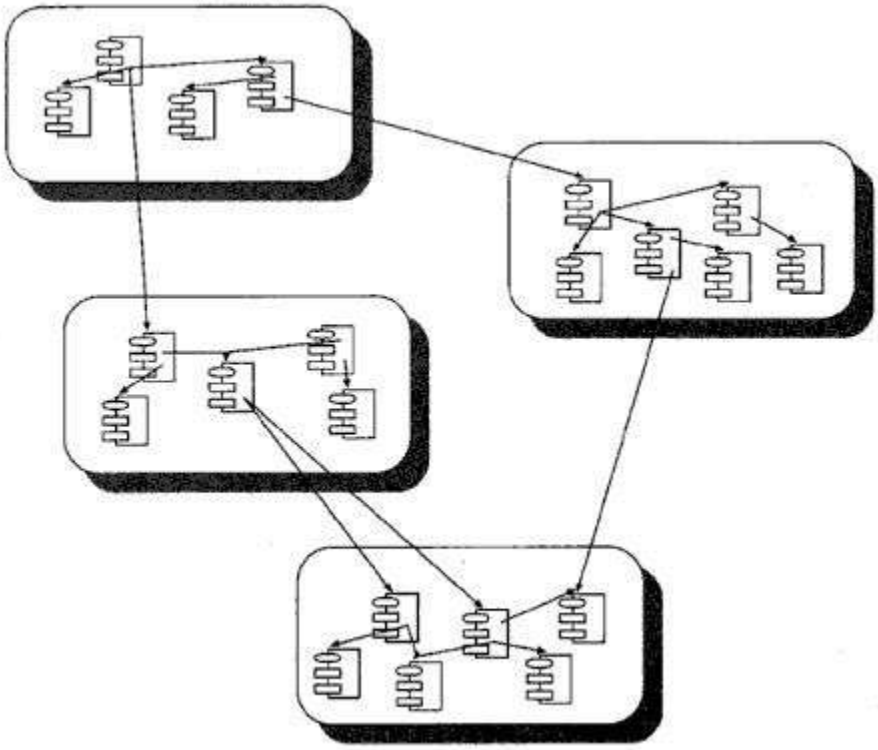


Figure 2-5
The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages

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.

Foundations of the Object Model

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

Actually, the object model has been influenced by a number of factors, not just object-oriented programming. Indeed, as the sidebar further discusses, the object model has proven to be a unifying concept in computer science, applicable not 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 upon proven ones. Unfortunately, most programmers today are formally and informally trained only in the principles of structured design. Certainly, many good engineers have developed and deployed countless useful software systems using these techniques. However, there are limits to the amount of complexity we can handle using only algorithmic decomposition; thus we must turn to object-oriented decomposition. Furthermore, if we try to use languages such as C++ and Ada as if they were only traditional, algorithmically oriented languages, we not only miss the power available to us, but we usually end up worse off than if we had used an older language such as C or Pascal. Give a power drill to a carpenter who knows nothing about electricity, and he would use it as a hammer. He will end up bending quite a few nails and smashing several fingers, for a power drill makes a lousy hammer.

OOP, OOD, and OOA

Because the object model derives from so many- disparate sources, it has unfortunately been accompanied by a muddle of terminology. A Smalltalk programmer uses *methods*, a C++ programmer uses *virtual member functions*, and a CLOS programmer uses *generic functions*. An Object Pascal programmer talks of a *type coercion*; an Ada programmer calls the same thing a *type conversion*. To minimize the confusion, let's define what is object-oriented and what is not. The glossary provides a summary of all the terms described here, plus many others.

Bhaskar has observed that the phrase object-oriented 'has been bandied about with carefree abandon with much the same reverence accorded 'rnotherhood,' 'apple pie,' and 'structured

programming' . What we can agree upon is that the concept of an object is central to anything object-oriented. In the previous chapter, we informally defined an object as a tangible entity that exhibits some well-defined behavior. Stefik and Bobrow define objects as "entities that combine the properties of procedures and data since they perform computations and save local state" . 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

Object-Oriented Programming What then, is object-oriented programming (or OOP, as it is sometimes written)? We define it as follows:

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

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

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

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

For a language to support inheritance means that it is possible to express "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, and CLOS are all object-oriented, and Ada is object-based. However, since objects and classes are elements of both kinds of languages, it is 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? We suggest that

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

There are two important parts to this definition: object-oriented design (1) leads to an object-oriented decomposition and (2) uses different notations to express different models of the logical (class and object structure) and physical (module and process architecture) design of a system, 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. We will occasionally use the acronym OOD to designate the particular method of object-oriented design described in this book.

Object-Oriented Analysis The object model has influenced even earlier phases of the software development life cycle. Traditional structured analysis techniques, best typified by the work of DeMarco , Yourdon , and Gane and Sarson , with real-time extensions by Ward, and Mellor and by Hatley and Pirbhai focus upon the flow of data within a system. Object-oriented analysis (or OOA, as it is sometimes called) emphasizes the building of real-world models, using an object-oriented view of the world:

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

How are OOA, OOD, and OOP related? Basically, the products of object oriented analysis 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.

Kinds of Programming Paradigms

Jenkins and Glasgow observe that "most programmers work in one language and use only one programming style. They program in a paradigm enforced by the language they use. Frequently, they have not been exposed to alternate ways of thinking about a problem, and hence have difficulty in seeing the ad-vantage of choosing a style more appropriate to the problem at hand,. 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]. They further suggest that there are five main kinds of programming styles, here listed with the kinds of abstractions they employ:

- Procedure-oriented — Algorithms
- Object-oriented — Classes and objects
- Logic-oriented — Goals, often expressed in a predicate calculus
- Rule-oriented — If-then rules
- Constraint-oriented — Invariant relationships

There is no single programming style that is best for all kinds of applications. For example, rule-oriented programming would be best for the design of a knowledge base, and procedure-oriented programming would be best suited 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 upon its own conceptual framework. Each requires a different mindset, a different way of thinking about the problem. For all things object-oriented, the conceptual framework is the object model. There are four major elements of this model:

Abstraction
Encapsulation
Modularity
Hierarchy

By *major*, we mean that a model without any one of these elements is not object-oriented.
There are three minor elements of the object model:

Typing
Concurrency
Persistence

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

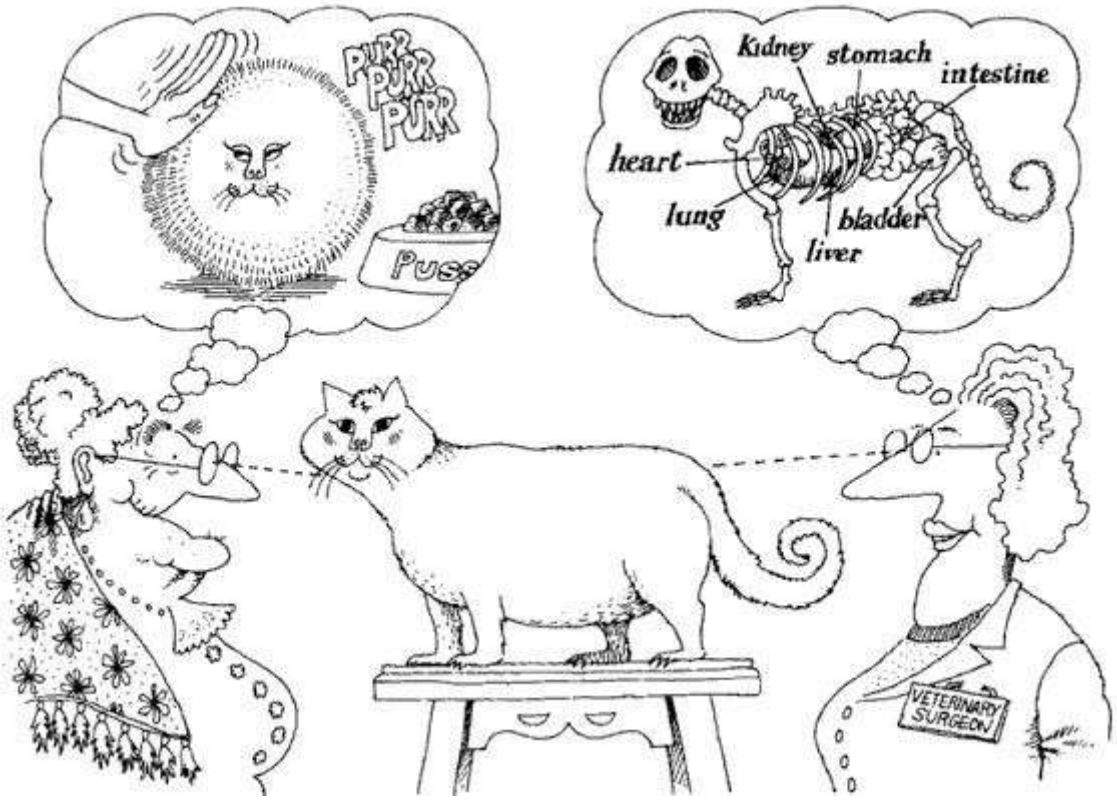
Without this conceptual framework, you may be programming in a language such as Smalltalk, Object Pascal, C++, CLOS, Eiffel, or Ada, but your design is going to smell like a FORTRAN, Pascal, or C application. You will have missed out on or otherwise abused the expressive power of the object-oriented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

Abstraction

The Meaning of Abstraction Abstraction is one of the fundamental ways that we as humans cope with complexity. Hoare suggests that "abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences" .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" .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", 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* achieved by applying the principle of least commitment, through which the interface of an object provides its essential behavior, and nothing more]. "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 beyond the scope of the abstraction.



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

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

Seidewitz and Stark suggest that "there is a spectrum of abstraction, from objects which closely model problem domain entities to objects which really have no reason for existence"
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 together operations that are all used by some superior level of control, or operations that all use some junior-level set of operations

Coincidental abstraction	An object that: packages a set of operations that have no relation to each other

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

A *client* is any object that uses the resources of another object (known as the *server*). We can characterize the behavior of an object by considering the services that it provides to other objects, as well as the operations that it may perform upon other objects. This view forces us to concentrate upon the outside view of an object, and leads us to what Meyer calls the *contract model* of programming : the outside view of each object defines a contract upon 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].

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 upon 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 *post conditions* (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. As we will describe later, certain languages permit objects to throw exceptions so as to abandon processing and alert some other object to the problem, who 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, and so we will use them interchangeably.

All abstractions have static as well as dynamic properties. For example, a file object takes up a certain amount of space on a particular memory device; it has a name, and it has contents.

These are all static properties. The value of each of these properties is dynamic, relative to the lifetime of the object: a file object may grow or shrink in size, its name may change, its contents may change. In a procedure-oriented style of programming, the activity that changes the dynamic value of objects is the central part of all programs: things happen when subprograms are called and statements are executed. In a rule-oriented style of programming, things happen when new events cause rules to fire, which in turn may trigger other rules, and so on. In an object-oriented style of programming, things happen whenever we operate upon an object (in Smalltalk terminology, when we *send a message* to an object). Thus, invoking an operation upon an object elicits some reaction from the object. What operations we can meaningfully perform upon an object and how that object reacts constitute the entire behavior of the object.

Examples of Abstraction Let's illustrate these concepts with some examples. Our purpose here is to show how we can concretely express abstractions, not, so much how we find the right abstractions for the given problem. We defer a complete treatment of this latter topic to Chapter 4.

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

One of the key abstractions in this problem is that of a sensor. Actually, there are several different kinds of sensors. Anything that affects production must be measured, and so we must have sensors for air and water temperature, humidity, light, pH, and nutrient concentrations, among other things. Viewed from the outside, 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 then is a location? It is some identifiable place on the farm at which we desire to measure the temperature; presumably, there are only a few such locations. What is important for a temperature sensor is not so much where it is located, but the fact that it has a unique location and identity 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 upon a temperature sensor? Our design decision is that a client can calibrate it, as well as ask what the current temperature is.

Let's use C++ to capture these design decisions. For those readers who are not familiar with C++, or for that matter any of the other object-oriented programming languages we mention

in this book, the appendix provides a brief overview of several languages, with examples. In C++, we might write the -4-1-rations that capture our abstraction of a temperature sensor:

```
//Temperature in degrees Fahrenheit
typedef float Temperature;

Number uniquely denoting the location of a sensor

typedef unsigned int Location;

class TemperatureSensor  {
Public:

TemperatureSensor(Location);
~TemperatureSensor() ;

void calibrate(Temperature actualTemperature);

Temperature currentTemperature() const;

private:
…
};
```

The two typedefs, **Temperature** and **Location**, provide convenient aliases for more primitive types, thus letting us express our abstractions in the vocabulary of the problem domain[5]. **Temperature** is a floating-point type representing temperature in degrees Fahrenheit. The type Location denotes the places where temperature sensors may be deployed throughout the farm.

The class **TemperatureSensor** captures our abstraction of a sensor itself; its representation is hidden in the private part of the class.

TemperatureSensor is defined as a class, not a concrete object, and therefore we must first create an *instance* so that we have something upon which to operate. For example, we might write:

```
Temperature temperature;

TemperatureSensor greenhouselSensor(l);
TemperatureSensor greenhouse2Sensor(2);

temperature = greenhouselSensor.currentTemperature();
```

Consider the invariants associated with the operation **currentTemperature**: its preconditions include the assumption that the sensor has been elaborated with a valid location, and its postconditions include the assumption that the value returned is in degrees Fahrenheit.

The abstraction we have described thus far is passive; some client object must operate upon an air temperature sensor object to determine its current temperature. However, there is another legitimate abstraction that may be more or less appropriate depending upon 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 upon but rather acts upon other objects whenever the temperature at its location changes a certain number of degrees from a given set point. 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? A common programming idiom used in such circumstances is the callback, in which a client provides a function to the server (the callback function), and the server calls the client's function whenever the appropriate conditions are met. Thus, we might write the following:

```
class ActiveTemperatureSensor {
public:

ActiveTemperatureSensor(Location, void (*f)(Location,

Temperature)); ~ActiveTemperatureSensor();

void calibrate(Temperature actualTemperature);
void establishSetpoint(Temperature setpoint, Temperature delta);

Temperature currentTemperature() const;

private:

…
};
```

This class is a bit more complicated than the first, but it captures our new abstraction quite well. Whenever we create a sensor object, we must as before provide its location, but we must now also provide a callback function whose signature includes a Location parameter and a **Temperature** parameter. Additionally, a client of this abstraction may invoke the operation **establishSetpoint** to establish a critical range of temperatures. It is then the responsibility of the **ActiveTemperatureSensor** object to invoke the given callback function whenever the temperature at its location drops below or rises above the given setpoint. When the callback is invoked, the sensor provides its location and the current temperature, so that the client has sufficient information to respond to the condition.

Notice that a client can still inquire as to the current temperature of a sensor at any time. What if a client never establishes a setpoint? Our abstraction must

make some reasonable assumption: one design decision might be to initially assume an infinite range of critical

temperatures, and so the callback would never be invoked until some client finally established a setpoint.

How the **ActiveTemperatureSensor** class 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' 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 factors should change over time to maximize the harvest. A growing plan is a legitimate entity abstraction, because it forms part of the vocabulary of the problem domain. Each crop has its own growing plan, -but the growing plans for all crops take the same form. Basically, a growing plan is a mapping of time to action. 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 65T 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.

A growing plan is thus responsible for keeping track of all interesting actions associated with growing a crop, correlated with the times at which those actions should take place. 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. 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.

From the perspective of the 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. For example, there might be an object that sits at the boundary of the human/machine interface and translates human input into plans. This is the object that establishes the details of a growing-plan, and so it must be able to change the state of a growing-plan object. There must also be an object that carries out the growing plan, and it must be able to read the details of a plan for a particular time.

As this example points out, no object stands alone; every object collaborates with other objects to achieve some behavior[6]. 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.

We might capture our design decisions for a growing plan as follows. First, we provide the following typedefs, so as to bring our abstractions closer to the Vocabulary of the problem domain:

```
Number denoting the day of the year
typedef unsigned int Day;
Number denoting the hour of the day
typedef unsigned int Hour;
   Boolean type
enum Lights {OFF, ON};
Number denoting acidity/alkalinity on a scale of 1 to 14
typedef float pH;
Number denoting percent concentration from 0 to 100
typedef float Concentration;
```

Next, as a tactical design decision, we provide the following structure:

```
Structure denoting relevant plan conditions
struct Condition {
  Temperature temperature;
  Lights lighting;
  pH acidity;
  Concentration concentration;
};
```

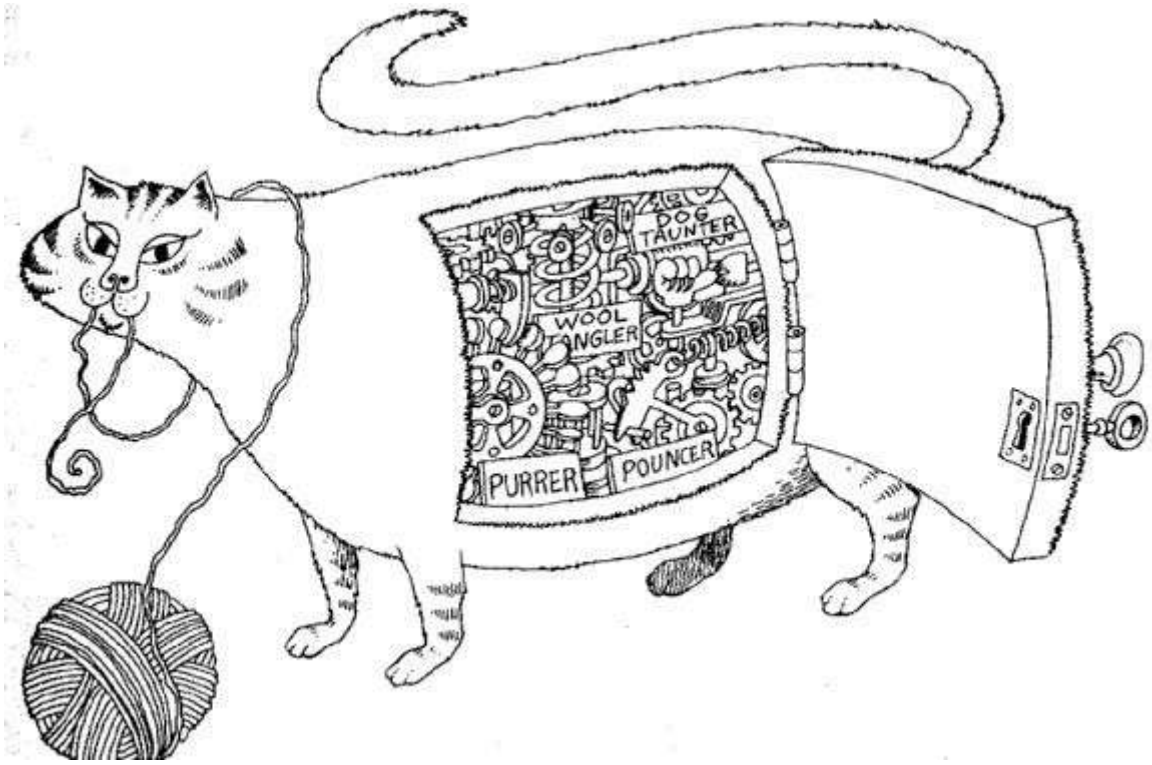
Here we have something less than an entity abstraction: a **Condition** is simply a physical aggregation of other things, with no intrinsic behavior. For this reason, we use a C++ record structure, rather than a C++ class, which has richer semantics.

Finally, we turn to the growing-plan class itself:

```
class GrowingPlan {
public:

GrowingPlan(char* name);
virtual ~GrowingPlan();

void clear();
virtual void establish(Day, Hour, const Condition&);

const char* name() const;
const Condition& desiredConditions(Day, Hour) const;

protected:
…
};
```

Notice that we have introduced one new responsibility to this abstraction: a growing plan has a name, which a client can set and inquire about. Also, note that we declare the operation **establish** as virtual, because we expect subclasses to override the default behavior provided by the class **GrowingPlan**.

In the declaration of this class, the public part exports *constructor* and *destructor member functions* (which provide for the birth and death of an object, respectively), two modifiers (the member functions clear and *establish*), and two *selectors* (the member functions *name* and *desiredConditions*). We have intentionally left out the private members (designated by the ellipses), because at this point in our design we wish to focus only upon the responsibilities of the class, not its representation.



Encapsulation hides the details of the implementation of an object.

The Meaning of Encapsulation Although we earlier described our abstraction of the class **GrowingPlan** 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 this class, 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. As Ingalls wisely suggests, "No part of a complex System should depend on the internal details of any other part.Whereas abstraction "helps people to think about what they- are doing," encapsulation allows program changes to be reliably made with limited effort" [51].

Abstraction and encapsulation are complementary concepts: abstraction focuses upon the observable behavior of an object, whereas *encapsulation* focuses upon the implementation that gives rise to this behavior. Encapsulation is most often achieved through *information biding*,

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.

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 upon 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.

Liskov goes as far as to suggest that "for abstraction to work, implementations must be encapsulated". 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.

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 upon this object: turn it on, turn it off, and find out if it is running. We do not make it a responsibility of this abstraction to maintain a fixed temperature. Instead, we choose to give this responsibility to another object, 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 upon 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 upon this separation of responsibilities, we make each individual abstraction more cohesive.

We begin with another **typedef**:

```
// Boolean type
enum Boolean {FALSE, TRUE};
```

For the heater class, in ' addition to the three operations mentioned earlier, we must also provide metaoperations, namely, constructor and destructor operations that initialize and destroy instances of this class, respectively. Because our system might have multiple heaters, we use the constructor to associate each software object with a physical heater, similar to the approach we used with the **TemperatureSensor** class. Given these design decisions, we might write the definition of the class *Heater* in C++ as follows:

```
Class Heater {
public:

Heater(location);
~Heater();

void turnOn();
void turnoff();

Boolean isOn() const;

private:
…
};
```

This interface represents all that a client needs to know about the class **Heater**.

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

Consider the following class, which captures our abstraction of a serial port:

```
Class SerialPort {
public:

SerialPort();
~SerialPort();

void write(char*);
void write(int);

static SerialPort ports[10];
```

```
private:
…
};
```

Here we provide a class whose instances denote actual serial ports, to which We can write strings and integers. Additionally, we declare an array of serial Ports, denoting all the different serial ports in our systems.

We complete the declaration of the class **Heater** by adding three attributes:

```
class Heater {
public:
…
protected:
  const Location repLocation;
  Boolean repIs0n;
  SerialPort* repPort;
};
```

These three attributes (**repLocation**, **repIsOn**, and **repPort**) form the encapsulated representation of this class. The rules of C++ are such that compiling client code that tries to access these member objects directly- will result in a semantic error.

We may next provide the implementation of each operation associated with this class:

```
Heater::Heater(location 1) : repLocation(1), repIs0n(FALSE),
repPort(&SerialPort::ports[1]) {}
Heater::~Heater() {}

void Heater::turnOn() {
  if (!repIs0n) {
    repPort->write(“*”);
    repPort->write(repLocation);
    repPort->write(1);
    repIs0n = TRUE;
  }
}

void Heater::turnoff() {
  if (repIs0n) {
    repPort->write(“*”);
    repPort->write(replocation);
    repPort->write(O);
    repIs0n = FALSE;
  }
}

Boolean Heater::isOn() const {
  return repIs0n;
```

```
}
```

This implementation is typical of well-structured object-oriented systems: the implementation of a particular class is generally small, because it can build upon the resources provided by lower-level classes.

Suppose that for whatever reason our system engineers choose to use memory-mapped I/0 instead of serial communication lines. We would not need to change the interface of this class; we would only need to modify its implementation. Because of C++'s obsolescence rules, we would probably have to recompile this class and the closure of its clients, but because the functional behavior of this abstraction would not change, we would not have to modify any code that used this class unless a particular client depended upon the time or space semantics of the original implementation (which would be highly undesirable and so very unlikely, in any case).

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 acceptable or that some objects consume more space than is available. In such situations, the representation of an object is often changed so that more efficient algorithms can be applied or so that one can optimize for space by calculating rather then storing certain data. This ability to change the representation of an abstraction without disturbing any of its clients is the essential benefit of encapsulation.

Ideally, attempts to access the underlying representation of an object should be detected at the time a client's code is compiled. How a particular language should address this matter is debated with great religious fervor in the object-oriented programming language community. For example, Smalltalk prevents a client from directly accessing the instance variables of another class; violations are detected at the time of compilation. On the other hand, Object Pascal does not encapsulate the representation of a class, so there is nothing in the language that prevents clients from referencing the fields of another object directly. CLOS takes an intermediate position; each slot may have one of the Slot options **:reader**, **:writer**, or **:accessor**,

which grant a client read access, write access, or read/write access, respectively. If none of these options are used, then the slot is fully encapsulated. By convention, revealing that some value is Stored in a slot is considered a breakdown of the abstraction, and so good CLOS style requires that when the interface to a class is published, only its generic function names are documented, and the fact that a slot has accessor functions is not revealed. C++ offers even more flexible control over the Visibility of member objects and member functions. Specifically, members may be placed in the public, private, or protected parts of a class. Members declared in the public parts are visible to all clients; members declared in the private parts are fully encapsulated; and members declared in the protected parts are visible only to the class itself and its subclasses. C++ also supports the notion of *friends*: cooperative classes that are permitted to see each other's private parts.

Hiding is a relative concept: what is hidden at one level of abstraction may represent the outside view at another level of abstraction. The underlying representation of an object can be revealed, but in most cases only if the creator of the abstraction explicitly exposes the implementation, and then only if the client is willing to accept the resulting additional complexity. Thus, encapsulation cannot stop a developer from doing stupid things: as Stroustrup points out, "Hiding is for the prevention of accidents, not the prevention of fraud"
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. In practice, there are times when one must study the implementation of a class to really understand its meaning, especially if the external documentation is lacking.

Modularity

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

Liskov states that "modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules. We will use the definition of Parnas: The connections between modules are the assumptions which the modules make about each other ". Most languages

that support the module as a separate concept also distinguish between the interface of a module and its implementation. Thus, it is

fair to say that modularity and encapsulation go hand in hand. As with encapsulation, particular languages support modularity in diverse ways. For example, modules in C++ are nothing more than separately compiled files. The traditional practice in the C/C++ community is to place module interfaces in files named with a h suffix; these are called *header files*. Module implementations are placed in files named with a c suffix[7]. Dependencies among files can then be asserted

Modularity packages abstractions into discrete units.

using the #**include** macro. This approach is entirely one of convention; it is neither required nor enforced by the language itself. Object Pascal is a little more formal about the matter. In this language, the syntax for units (its name for modules) distinguishes between module interface and implementation. Dependencies among units may be asserted only in a module's interface. Ada goes one step further. A package (its name for modules) has two parts: the package specification and the package body. Unlike Object Pascal, Ada allows connections among modules to be asserted separately in the specification and body of a package. Thus, it is possible for a package body to depend upon modules that are otherwise not visible to the package's specification.

Deciding upon the right set of modules for a given problem is almost as hard a problem as deciding upon the right set of abstractions. Zelkowitz is absolutely right when he states that "because the solution may not be known *hen 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".

Modules serve as the physical containers in which we declare the classes and objects of our logical design. This is no different than the situation faced by the electrical engineer designing a board-level computer. NAND, NOR, and NOT gates might be used to construct the necessary logic, but these gates must be physically- packaged in standard integrated circuits, such as a 7400, 7402, or 7404. Lacking any such standard software parts, the software engineer has considerably more degrees of freedom - as if the electrical engineer had a silicon foundry at his or her disposal.

For tiny problems, the developer might decide to declare every class and object in the same package. For anything but the most trivial software, a better solution is to group logically related classes and objects in the same module, and expose only those elements that other modules absolutely must see. This kind of modularization is a good thing, but it can be taken to extremes. For example, consider an application that runs on a distributed set of processors and uses a message passing mechanism to coordinate the activities of different programs. in a large system, like that described in Chapter 12, it is common to have several hundred or even a few thousand kinds of messages. A naive strategy might be to define each message class in its own module. As it turns out, this is a singularly poor design decision. Not only does it create a documentation nightmare, but it makes it terribly difficult for any users to find the classes they need. Furthermore, when decisions change, hundreds of modules must be modified or recompiled. This example shows how information hiding can backfire. Arbitrary modularization is sometimes worse than no modularization at all.

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

Our experience indicates that there are several useful technical as well as nontechnical guidelines that can help us achieve an intelligent modularization of classes and objects. As Britton and Parnas have observed, "The overall goal of the decomposition into modules is the reduction of software cost by allowing modules to be designed and revised independently. . .
Each module's structure should be simple enough that it can be understood fully; it should be possible to change the implementation of other modules without knowledge of the implementation of other modules and without affecting the behavior of other modules; [and] the case of making a change in the design should bear a reasonable relationship to the likelihood of the change being needed". 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

upon this interface, the modules that depend upon these modules, and so on. Thus, for very large programs (assuming that

our development environment does not support incremental compilation), a change in a single module interface might result in many minutes if not hours of recompilation. 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 all using modules. Our style is to hide as much as we can in the implementation of a module incrementally shifting declarations from a modules implementation to its interface is far less painful and destabilizing than ripping out extraneous interface code.

The developer must therefore balance two competing technical concerns: the desire to encapsulate abstractions, and the need to make certain abstractions visible to other modules. Parnas, Ciements, and Weiss offer the following guidance: "System details that are likely to change independently should be the secrets of separate modules; the only assumptions that should appear between modules are those that are considered unlikely to change. Every data structure is private to one module; it may be directly accessed by one or more programs within the module but not by programs outside the module. Any other program that requires information stored in a module's data Structures must obtain it by calling module programs"
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 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 at 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 no technical needs may also affect modularization decisions. Typically, work assignments in a development team are given on a Module-by-module basis, and so the boundaries of modules may be established to minimize the interfaces among different parts of the development organization. Senior designers are usually given responsibility for module -Interfaces, and more junior developers complete their implementation. On a larger scale, the same situation

applies with subcontractor relationships. Abstractions may be packaged so as to quickly stabilize the module interfaces agreed upon among the various

companies. Changing such interfaces usually involves much wailing and gnashing of teeth - not to mention a vast amount of paperwork - and so this factor often leads to conservatively designed interfaces. Speaking of paperwork, modules also usually serve as the unit of documentation and configuration management. Having ten modules where one would do 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 that instead of building some special-purpose hardware, we decide to use a commercially available workstation, and employ an off-the-shelf graphical user interface (GUI). 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. In C++, we might write the header file for this module (which we name **gplan.h**) as:

```
// gplan.h

#ifndef _GPLAN_H
#define _GPLAN_H 1

#include "gtypes.h"
#include "except.h"
#include "actions.h"

class GrowingPlan ...

class FruitGrowingPlan ...

class GrainGrowingPlan ...

…
#endif
```

Here we import three other header files (**gtypes.h**, **except.h**, and **actions.h**), upon whose interface we must rely.

The implementations of these growing-plan classes then appear in the implementation of this module, in a file we name (by convention) **gplan.cpp**.

We might also define a module whose purpose is to collect all of the code associated with application-specific dialog boxes. This unit most likely depends upon the classes declared in the interface of **gplan.h**, as well as files that encapsulate certain GUI interfaces, and so it must in turn include the header file gplan.h, as well as the appropriate GUI header files.

Our design will probably include many other modules, each of which imports the interface of lower level units. Ultimately, we must define some main program from which we can invoke this application from the operating ,system. In object-oriented design, defining this main program is often the least important decision, whereas in traditional structured design, the main program serves as the root, the keystone that holds everything else together. We suggest hat 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 fields rather artificial answers.... Real systems have no top".

Hierarchy

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

We define hierarchy as follows:

> *Hierarchy is a ranking or ordering of abstractions.*

The most important hierarchies in a complex system are its class structure e "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 systems. Basically, inheritance defines a relationship among classes, 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.

Semantically, inheritance denotes an "is-a" relationship. For example, a bear a" kind of mammal a house "is a" kind of tangible asset, and a quick sort "is sorting algorithm. Inheritance thus implies a generalization/specialization hierarchy, wherein a subclass

specializes the more general structure or behavior of its superclasses. Indeed, this is the litmus test r 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 generalized growing plan. Different kinds of crops, however, demand specialized growing plans. For example, the growing plan for all fruits is

Abstractions form a hierarchy.

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 specialized behavior common to all fruits, such as the knowledge of when to pollinate or when to harvest the fruit. We can assert this "is a" relationship among these abstractions in C++ as follows:

```
// yield type
typedef unsigned int Yield;

class FruitGrowingPlan : public GrowingPlan {

public:

  FruitGrowinjgPlan(char* name);
  virtual ~FruitGrowingPlan();

  virtual void establish(Day, Hour,

  Condition&); void scheduleHarvest(Day, Hour);

  Boolean isHarvested() const;
  unsigned daysUntilHarvest() const;
  Yield estimatedYield() const;

protected:

  Boolean repHarvested;
  Yield repYield;
};
```

This class declaration captures our design decision wherein a **FruitGrowingPlan** "is -a" kind of **GrowingPlan**, with some additional structure (the member objects **repHarvested** and **repYield**) and behavior (the four new member functions, plus the overriding of the superclass operation establish). Using this class, we could declare even more specialized subclasses, such as the class **AppleGrowingPlan** .

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 . As Cox points out, «Without inheritance, every class would be a free-standing unit, each developed from the ground up. Different classes would bear no relationship with one another, since the developer of each provides methods in whatever manner he Chooses. Any consistency across classes is the result of discipline on the part of the programmers. Inheritance makes it possible to define new software in the Same way we introduce any concept to a newcomer, by comparing it with something that is already familiar".

There is a healthy tension among the principles of abstraction, encapsulation, and hierarchy. As Danforth and Tomlinson point out, "Data abstraction attempts to provide an opaque barrier behind which methods and state are hidden; inheritance requires opening this interface to some extent and may allow state as well as methods to be accessed without abstraction". For a given class, there are usually two kinds of clients: objects that invoke

operations upon instances of the class, and subclasses that inherit from the class. Liskov therefore notes that, with inheritance, encapsulation can be violated in one of three ways: "The subclass might access an instance variable of its superclass, call a private operation of its superclass, or refer directly to superclasses of its superclass". Different programming languages trade off support for encapsulation and inheritance in different ways, but among the languages described in this book, C++ offers perhaps the greatest 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. in C++, we might declare this class as follows:

```
class Plant {
public:

  Plant (char* name, char* species);
  virtual ~Plant();

  void setDatePlanted(Day);
  virtual establishGrowingConditions(const Condition&);

  const char* name() const;
  const char* species() const;
  Day datePlanted() Const;

Protected:
  char* repName;
  char* repSpecies;
  Day repPlanted;

private:
…
};
```

According to this class definition, each instance of the class Plant has a name, species, and date of planting. Additionally, optimal growing conditions may be established for each particular kind of plant. Because we expect this behavior to be specialized by subclasses, we declare this operation as *virtual* in C++[8]. Notice that the three member objects are declared as protected; thus, they are accessible only to the class itself and its subclasses. On the other hand, all members declared in the private part are accessible only to the class itself.

Our analysis of the problem domain might suggest that flowering plants 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, a **FlowerFruitVegetable**, that duplicated information from the
Flower and **FruitVegetablePlant** 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 fruits and vegetables:

```
class FlowerMixin {
public:

  FlowerMixin(Day timeToFlower, Day

  timeToSeed); virtual ~FlowerMixin();

  Day timeToFlower() const;
  Day timeToSced() const;

protected:
…
};

class FruitVegetableMixin {
public:

  FruitVegetablieMixin(Day timeToHarvest);

  virtual ~FruitVegetableMixin();

  Day timeToHarvesto const;

protected:

…
};
```

Notice that these two classes have no superclass; they standalone. 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 as follows:

```
class Rose : public Plant, public FlowerMixin...
```

Similarly, a Carrot class can be declared as follows:

```
class Carrot : public Plant, public FruitVegetableMixin {};
```

in both cases, we form the subclass by inheriting from two superclasses. 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**. Now, suppose we want to declare a class for a plant such as the cherry tree that has both flowers and fruit. We might write the following:

```
class Cherry :  public Plant,
     public FlowerMixin,
     public FruitVegetableMixin ...
```

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

Multiple inheritance is often overused. For example, cotton candy is a kind of candy, but it is distinctly not a kind of cotton. Again, the litmus test for inheritance applies: if B is not a kind of A, then B should not inherit from A. Often, i11-formed multiple inheritance lattices can be reduced to a single superclass plus aggregation of the other classes by the subclass.

Examples of Hierarchy: Aggregation Whereas these "is a" hierarchies denote generalization/specialization relationships, "part of, hierarchies describe aggregation relationships. For example, consider the following class:

```
class Garden {
public:

  Garden();
  virtual ~Garden();

protected:

  Plant* repPlants[100];
  GrowingPlan repPlan;
};
```

Here we have the abstraction of a garden, consisting of a collection of plants together with a growing plan.

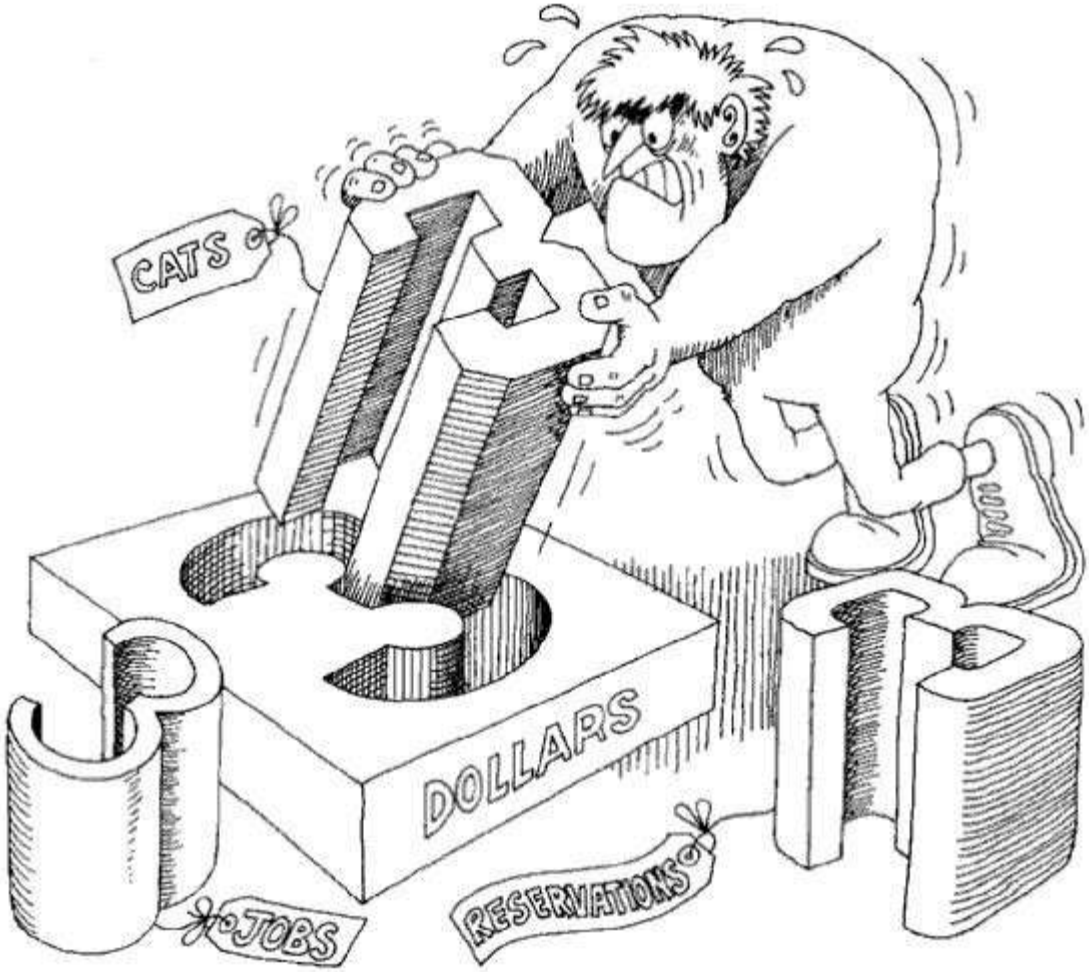
When dealing with hierarchies such as these, we often speak of *levels* of *abstraction*, a concept first described by Dijkstra. In terms of its "is a" hierarchy, a high-level abstraction is generalized, and a low-level abstraction is specialized. Therefore, we say that a **Flower** class is at a higher level of abstraction then 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, upon which it builds.

Aggregation is not a concept unique to 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 one different abstractions.

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: We capture this design decision in the example above, by including pointers to Plant objects rather than values. In contrast, we have decided that a **GrowingPlan** object is intrinsically associated with a **Garden** object, and does not exist independently of the garden. For this reason, we use a value of **GrowingPlan**. 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. We will discuss the semantics of ownership by value versus reference more detail in the next chapter.

Typing

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". For our purposes, we will use the terms type and class interchangeably[9]. Although the concepts of a *type* and a *class* are similar, we include typing as a separate element of the object model because the concept of a type places a very different emphasis upon the meaning of abstraction. Specifically, we state the following:



Strong typing prevents mixing abstractions.

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

Typing lets us express our abstractions so that the programming language in which we implement them can be made to enforce design decisions. Wegner observes that this kind of enforcement is essential for programming-in-the-large . The idea of conformance is central to the notion of typing. For example, consider units of measurement in physics. When we divide distance by time, we expect some value denoting speed, not weight. Similarly, multiplying temperature by a unit of force doesn't make sense, but multiplying mass by force does. These are both examples of strong typing, wherein the rules of our domain prescribe and enforce certain legal combinations of abstractions.

Examples of Typing: Strong and Weak Typing 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 upon an object unless the exact signature of that operation is defined in the object's class or superclasses. In strongly typed languages, violation of type

conformance can be detected at the time of compilation. Smalltalk, on the other hand, is an untyped language: a client can send any message to any class (although a class may not know how respond to the message). Violations of type conformance may not be known until execution, and usually manifest themselves as execution errors. Languages such as C++ are hybrid: they have tendencies toward strong typing, but it is possible to ignore or suppress the typing rules.

Consider the abstraction of the various kinds of storage tanks that might exist in a greenhouse. We are likely to have storage tanks for water as well as various nutrients; although one holds a liquid and the other a solid, these abstractions are sufficiently similar to warrant a hierarchy of classes, as the following example illustrates. First, we introduce another typedef:

```
Number denoting level from 0 to 100 percent
typedef float Level;
```

C++, typedefs do not introduce new types. In particular, the typedefs **Level** and Concentration are both floating-point numbers, and can be intermixed. In this aspect, C++ is weakly typed: values of primitive types such as int and **float** are indistinguishable within that particular type. In contrast, languages such as Ada and Object Pascal enforce strong typing among primitive types. In Ada, for example, the derived type and subtype constructs allow the developer to define distinct types, constrained by range or precision from more general types.

Next, we have the class hierarchy for storage tanks:

```
class StorageTank {
public:

  StorageTank();
  virtual ~StorageTank();

  virtual void fill();
  virtual void startDraining();
  virtual void stopOraining();

  Boolean isEmpty() const;
  Level level() const;

protected:
…
};

class WaterTank : public  StorageTank {
public:

  WaterTank();
  ~WaterTank();

  virtual void fill();
```

```
  virtual void startDraining();
  virtual Void stopDraining();
  void startHeating();
  void stopHeating();

  Temperature currentTemperature() const;
Protected:
…
};

class NutrientTank : public StorageTank
{ public:
  NutrientTank();
  virtual ~NutrientTank();

  virtual void startDraining();
   virtual void stopDraining();

Protected:
…
};
```

The class **StorageTank** is the base class in this hierarchy, and provides the structure and behavior common to all such tanks, such as the ability to fill and drain the tank. **WaterTank** and **NutrientTank** are both subclasses of **StorageTank**. Both subclasses redefine some of the behavior of the superclass, and the class **WaterTank** introduces some new behavior associated with temperature.

Suppose that we have the following declarations:

```
StorageTank s1, s2;
WaterTank w;
NutrientTank n;
```

Variables such as **s1**, **s2**, **w**, and n are not objects. To be precise, these are simply names we use to designate objects of their respective classes: when we say "the object **s1**," we really mean the instance of **StorageTank** denoted by the variable **s1**. We will explain this subtlety again in the next chapter.

With regard to type checking among classes, C++ is more strongly typed, meaning that expressions that invoke operations are checked for type correctness at the time of compilation. For example, the following statements are legal:

```
Level 1 = sl.level();
w.startDraining();
n.stopDraining();
```

In the first statement, we invoke the selector **level**, declared for the base class **StorageTank**. In the next two statements, we invoke a modifier (**startDraining**, and **stopDraining**) declared in the base class, but overridden in the subclass.

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

```
sl.startHeatinI();    Illegal
n.stopHeating();      Illegal
```

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

```
n.fill();
```

though fill is not defined in the class **NutrientTank** it is defined in the superclass **StorageTank**, from which the class **NutrientTank** inherits its structure and behavior.

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. Also, in the absence of parameterized classes, which we will discuss further in the next chapter and in Chapter 9, it is problematic to have type-safe collections of heterogeneous objects. For example, suppose we need the abstraction of a greenhouse inventory, which collects all of the tangible assets associated with a particular greenhouse. A common C idiom applied to C++ is to use a container class that stores pointers to void, which represents objects of an indefinite type:

```
class Inventory {
public:

  Inventory()
  ~Inventory();

  void add(void*);
  void remove(void*);

  void* mostRecento const;

  void apply(Boolean (*)(void*));
private:
…
};
```

The operation apply is an iterator, which allows us to apply an operation to every item in the collection. We will discuss iterators in more detail in the next chapter.

Given an instance of the class **Inventory**, we may add and remove pointers to objects of any class. However, this approach is not type-safe: we can legally add tangible assets such as storage tanks to an inventory, as well as nontangible assets, such as temperature or growing plans, which violates our abstraction of an inventory. Similarly, we might add a **WaterTank** object as well as a **TemperatureSensor** object, and unless we are careful, invoke the selector **mostRecent**, expecting to find a water tank when we are actually returned a storage tank.

There are two general solutions to these problems. First, we could use a type-safe container class. Instead of manipulating pointers to void, we might define an inventory class that manipulates only objects of the class **TangibleAsset**, which we would use as a mixin class for all classes that represent tangible assets, such as **WaterTank** but not **GrowingPlan**. 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 Smalltalk, for example, it is possible to query an object for its class. In C++, runtime type identification is not yet part of the language standard[10], but a similar effect can be achieved pragmatically, by defining an operation in the base class that returns a string or enumeration type identifying the particular class of the object. 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.

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

```
s1 = s2;
S1 = w;
```

The first statement is legal because the class of the variable on the left side of the statement (**StorageTank**) is the same as the class of the expression on the right side. The second statement is also legal because the class of the variable on the left side (**StorageTank**) is a superclass of the variable on the right side (**WaterTank**). However, this assignment results in a loss of information (known in C++ as *ilicing*). The subclass WaterTank introduces structure and behavior beyond that defined in the base class, and this information cannot be copied to an instance of the base class.

Consider the following illegal statements:

```
w = s1; // Illegal
w = n; // Illegal
```

The first statement is not legal because the class of the variable on the left side of the assignment statement (**WaterTank**) is a subclass of the class of the variable on the right side

(**StorageTank**). The second statement is illegal because the classes of the two variables are peers, and are not along the same line o inheritance (although they have a common superclass).

In some situations, it is necessary to convert a value from one type to another. For example, consider the following function:

```
void checkLevel(const StorageTank& s);
```

lf and only if we are certain that the actual argument we are given is of the class **WaterTank**, then we may explicitly coerce the value of the base class to the subclass, as in the following expression:

```
if «(WaterTank&)s).currentTemperature0 < 32.0) ...
```

This expression is type-consistent, although it is not completely type-safe. For example, if the variable s happened to denote an object of the class **NutrientTank** at runtime, then the coercion would fail with unpredictable results during execution. In general, type conversion is to be avoided, because it often represents a violation of abstraction.

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".

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

Examples of Typing: Static and Dynamic Binding The concepts of strong typing d static typing are entirely different. Strong typing refers to type consistency, whereas static typing - also known as *static binding* or *early binding* - refers Po the time when names are bound to types. Static binding means that the types all variables and expressions are fixed at the time of compilation; *dynamic binding* (also called late *binding*) means that the types of all variables and expressions are not known until runtime. Because strong typing and binding independent concepts, a language may be both strongly and statically typed strongly typed yet support dynamic binding (Object Pascal and C++), or untyped yet support dynamic binding

(Smalltalk). CLOS fits somewhere between C++ and Smalltalk, in that an implementation may either enforce or ignore any type declarations asserted by a programmer.

Let's again illustrate these concepts with an example from C++. Consider the following nonmember function[11]:

```
void balanceLevels(StorageTank& s1, StorageTank& s2);
```

Calling the operation **balanceLevels** with instances of **StorageTank** or any of its subclasses is type-consistent because the type of each actual parameter is part of the same line of inheritance, whose base class is **StorageTank.**

In the implementation of this function, we might have the expression:

```
if (s1.level() > s2.level())
  s2.fill();
```

What are the semantics of invoking the selector **level**? This operation is declared only in the base **StorageTank**, and therefore, no matter what specific class or subclass instance we provide for the formal argument **s1**, the base class operation will be invoked. Here, the call to level is statically bound: at the time of compilation, we know exactly what operation will be invoked.

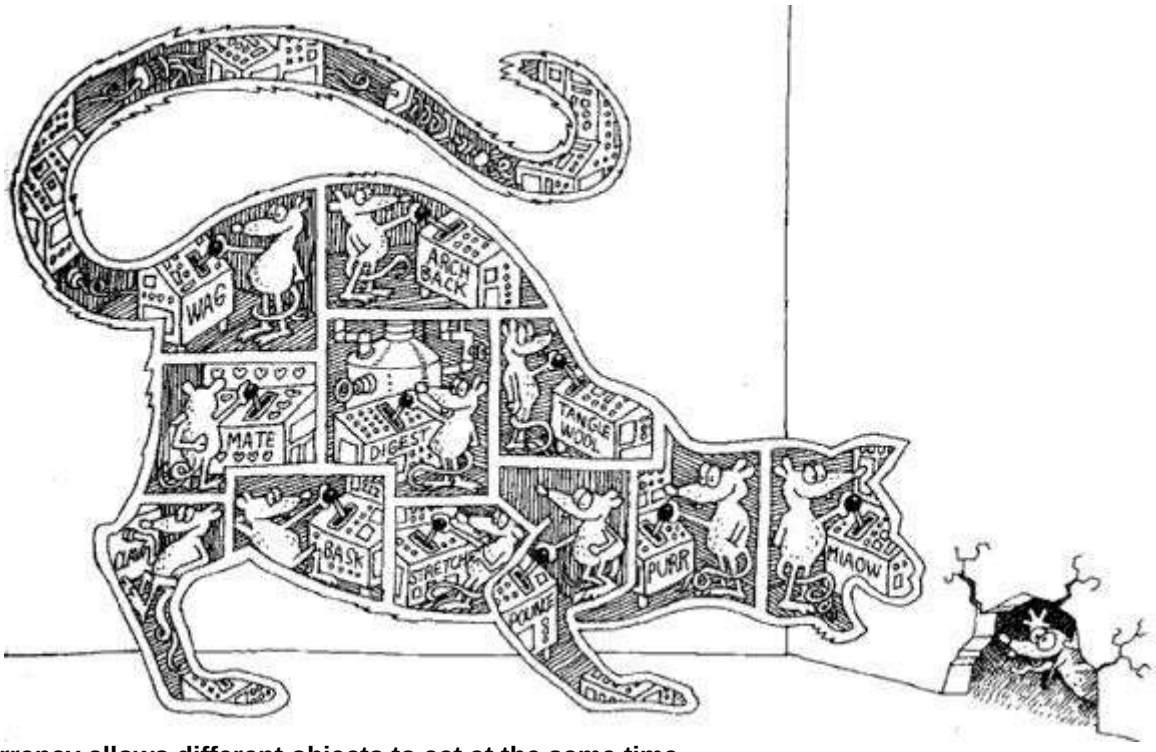
On the other hand, consider the semantics of invoking the modifier **fill**, which is dynamically bound. This operation is declared in the base class and then redefined only in the subclass **WaterTank**. If the actual argument to **s1** is a **WaterTank** instance, then **WaterTank::fill** will be invoked; if the actual argument to **s1** is a **NutrientTank** instance, then **StorageTank::fill** will be invoked[12].

This feature is called *polymorpbism*; it represents a concept in type theory in which a single name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass. Any object denoted by this name is therefore able to respond to some cormnon set of operations. The opposite of polymorphism is *monomorpbism*, which is found in all languages that are both strongly typed and statically bound, such as Ada.

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

Concurrency

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



Concurrency allows different objects to act at the same time.

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.

Many contemporary operating systems now provide direct support for currency, and so there is greater opportunity (and demand) for concurrency in object-oriented systems. For example, UNIX provides the system call *fork*, which spans a new process. Similarly, Windows/NT and OS/2 are multithreaded, and provide programmatic interfaces for creating and manipulating procces.

Lim and Johnson point out that "designing features for concurrency in OOP ages is not much different from [doing so in] other kinds of languages-concurrency is orthogonal to OOP at the lowest levels of abstraction. OOP or not, all the traditional problems in concurrent programming still remain". Indeed, building a large piece of software is hard enough; designing one that encompasses multiple threads of control is much harder because one must worry about such issues as deadlock, livelock, starvation, mutual exclusion and race conditions. Fortunately, as Lim and Johnson also point out, "At the highest levels of abstraction, OOP can alleviate the concurrency problem for the majority of programmers by hiding the concurrency inside reusable abstractions". Black *et al*. therefore suggest that "an object model is appropriate for a distributed system because it implicifly defines (1) the units of distribution and movement and (2) the entities that communicate".

Whereas object-oriented programming focuses upon data abstraction, encapsulation, and inheritance, concurrency focuses upon process abstraction and synchronization. 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 tbe properly that distinguisbes an active object from one tbat is not active.*

Examples of Concurrency Our carlier discussion of abstraction introduced the class **ActiveTemperatureSensor**, whose behavior required periodically sensing the current temperature and then invoking the callback function of a client object whenever the temperature changed a certain number of degrees from a given setpoint. We did not explain how the class implemented 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. For example, Ada's mechanism for expressing a concurrent process is the task. Similarly, Smalltalk provides the class **Process**, which we may use as the superclass of all active objects. There are a number of other concurrent object-oriented programming languages, such as Actors, Orient 84/K, and ABCL/1, that provide similar mechanisms for concurrency and synchronization.

In cach 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. This is the approach taken by the AT&T task library for C++, which provides the classes **Sched**, **Timer**, **Task**, and others. Naturally, the implementation of this library is highly platform-dependent, although the interface to the library is relatively portable. In this approach, concurrency is not an intrinsic part of the language (and so does not place any burdens upon 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, 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 upon is not corrupted when both active objects try to update its state simultaneously. This is the point where the ideas of abstraction, encapsulation, and concurrency interact. In the presence of concurrency, it is not enough simply to define the methods of an object; we must also make certain that the semantics of these methods are preserved in the presence of multiple threads of control.

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"

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 object.

UnifVing the concepts of concurrency and objects gives rise to concurrent object-oriented programming languages. In a similar fashion, introducing the concept of persistence to the object model gives rise to object-oriented databases. In practice, such databases build upon proven technology, such as sequential, indexed, hierarchical, network, or relational database models, but then offer to the programmer the abstraction of an object-oriented interface, through which database queries and other operations are completed in terms of objects whose lifetime transcends the lifetime of an individual program. This unification vastly simplifies the development of certain kinds of applications. In particular, it allows us to apply the same design methods to the database and nondatabase segments of an application, as we will see in Chapter 10.

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

Very few object-oriented programming languages provide direct support for persistence; Smalltalk is one notable exception, wherein there are protocols for streaming objects to and from disk (which must be redefined by subclasses). However, streaming objects to flat files is a naive solution to persistence that does not scale well. More commonly, persistence is achieved through a modest number of commercially available object-oriented databases . Another reasonable approach to persistence is to provide an object-oriented skin over a relational database. 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.

Applylng the Object Model

Benefits of 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 upon these earlier models. Thus, the object model offers a number of significant benefits that other models simply do not provide. Most importantly, the use of the object model leads us to construct systems that embody the five attributes of well-structured complex systems. In our experience, there are five other practical benefits to be derived from the application of the object model.

First, the use of the object model helps us to exploit the expressive power of object-based and object-oriented programming languages. As Stroustrup points out, "lt 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" . Our experience has been that, without the application of the elements of the object model, the more powerful features of languages such as Smalltalk, Object Pascal, C++, CLOS, and Ada are either ignored or greatly misused.

Next, the use of the object model encourages the reuse not only of software but of entire designs, leading to the creation of reusable application frame-works [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.

Third, the use of the object model produces systems that are built upon 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.

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

Figure 2-6
Applications of the Object Model

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

Applications of the Object Model

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

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