Chapter 1

# Complexity

A physician, a civil engineer, and a computer scientist were arguing about what was the oldest profession in the world. The physician remarked, "Well, in the Bible, it says that God created Eve from a rib taken out of Adam. This clearly required surgery, and so I can rightly claim that mine is the oldest profession in the world." The civil engineer interrupted, and said, "But even earlier in the book of Genesis, it states that God created the order of the heavens and the earth from out of the chaos. This was the first and certainly the most spectacular application of civil engineering. Therefore, fair doctor, you are wrong: mine is the oldest profession in the world." The computer scientist leaned back in her chair, smiled, and then said confidently, "Ah, but who do you think created the chaos?"

"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. Doing that 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 development 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.

# 1.1 The Structure of Complex Systems

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. Many even more complex systems appear in nature, such as the human circulatory system and the structure of a habanero pepper plant.

## The Structure of a Personal Computer

A personal computer is a device of moderate complexity. Most 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 CD or DVD drive and 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). Each of these has a different, specific 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 life. 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. Stars, planets, and 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.

## 1.2 The Inherent Complexity of Software

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" [1].

### Defining Software 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 on 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.

## 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 development process, the flexibility possible through software, and the problems of characterizing the behavior of discrete systems.

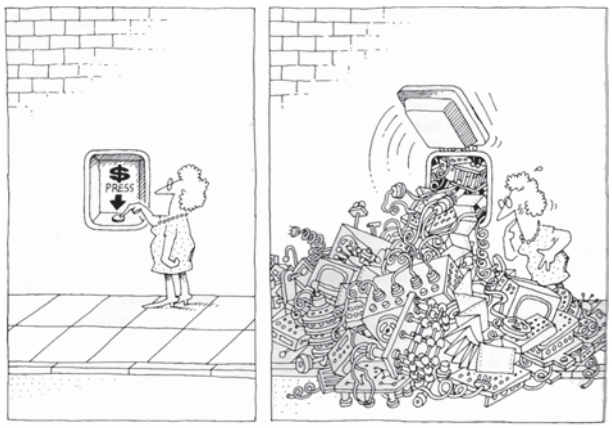
### *The Complexity of the Problem Domain*

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 "communication gap" 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 some 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 to express requirements 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 corners of a system's desired behavior.




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

Because a large software system is a capital investment, we cannot afford to scrap an existing system every time its requirements change. Planned or not, systems tend to evolve over time, a condition that is often incorrectly labeled software maintenance. To be more precise, it is *maintenance* when we correct errors; it is *evolution* when we respond to changing requirements; it is *preservation* when we continue to use extraordinary means to keep an ancient and decaying piece of software in operation. Unfortunately, reality suggests that an inordinate percentage of software development resources are spent on software preservation.

### The Difficulty of Managing the Development Process

The fundamental task of the software development team is to 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 frameworks of existing designs and code. However, the sheer volume of a system's requirements is sometimes inescapable and forces us either to write a large amount of new software or to reuse existing software in novel ways. Just a few 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. Having more developers means more complex communication and hence more difficult coordination, particularly if the team is geographically dispersed, as is often the case. 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 onsite 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 on

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.[1] 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 our 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, "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 on 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. When a ship's propulsion

1. Actually, even simple continuous systems can exhibit very complex behavior because of the presence of chaos. Chaos introduces a randomness that makes it impossible to precisely predict the future state of a system. For example, given the initial state of two drops of water at the top of a stream, we cannot predict exactly where they will be relative to one another at the bottom of the stream. Chaos has been found in systems as diverse as the weather, chemical reactions, biological systems, and even computer networks. Fortunately, there appears to be underlying order in all chaotic systems, in the form of patterns called attractors.

system fails due to a mathematical overflow, which in turn was caused by someone entering bad data in a maintenance system (a real incident), we understand the seriousness of this issue. There has been a dramatic rise in software-related system failures in subway systems, automobiles, satellites, air traffic control systems, inventory systems, and so forth. 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.

# 1.3 The Five Attributes of a Complex System

Considering the nature of this complexity, we conclude that there are five attributes common to all complex systems.

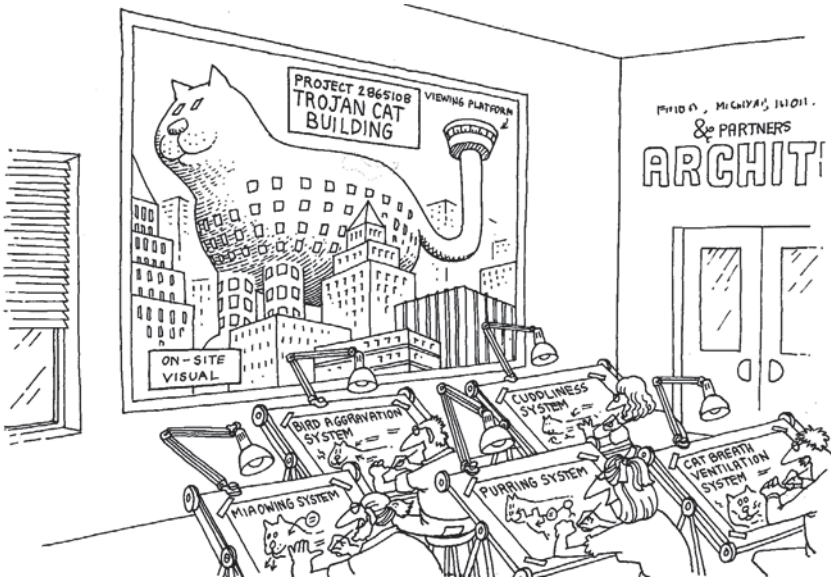
## Hierarchic Structure

Building on 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. "All systems have subsystems and all systems are parts of larger systems. . . . The value added by a system must come from the relationships between the parts, not from the parts per se" [9].




The architecture of a complex system is a function of its components as well as the hierarchic relationships among these components.

## Relative Primitives

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.

## Separation of Concerns

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 intercomponent 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.

## Common Patterns

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.

## Stable Intermediate Forms

Earlier, we noted that complex systems tend to evolve over time. Specifically, "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:

> 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 on 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.

# 1.4 Organized and Disorganized Complexity

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 learn how to fly a similar one.

## The Canonical Form of a Complex System

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.

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* of the system, respectively.[2]

For those of you who are familiar with object technology, let us be clear. In this case, where we are speaking of class structure and object structure, we are not referring to the classes and objects you create when coding your software. We are referring to classes and objects, at a higher level of abstraction, that make up complex systems, for example, a jet engine, an airframe, the various types of seats, an autopilot subsystem, and so forth. You will recall from the earlier discussion on the attributes of a complex system that whatever is considered primitive is relative to the observer.

In Figure 1–1 we see the two orthogonal hierarchies of the system: its class structure and its object structure. Each hierarchy is layered, with the more abstract

2. Complex software systems embody other kinds of hierarchies as well. Of particular importance is the module structure, which describes the relationships among the physical components of the system, and the process hierarchy, which describes the relationships among the system's dynamic components.

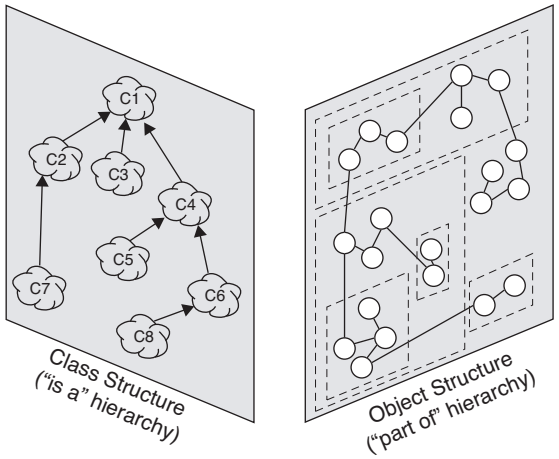



**Figure 1–1** The Key Hierarchies of Complex Systems

classes and objects built on more primitive ones. What class or object is chosen as primitive is relative to the problem at hand. Looking inside any given level reveals yet another level of complexity. Especially among the parts of the object structure, there are close collaborations among objects at the same level of abstraction.

Combining the concept of the class and object structures together with the five attributes of a complex system (hierarchy, relative primitives [i.e., multiple levels of abstraction], separation of concerns, patterns, and stable intermediate forms), we find that virtually all complex systems take on the same (canonical) form, as we show in Figure 1–2. Collectively, we speak of the class and object structures of a system as its *architecture*.

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. (In Figure 1–2, note classes C3, C5, C7, and C8 and the number of the instances 03, 05, 07, and 08.) As the figure suggests, there are usually many more objects than classes of objects within a complex system. By showing the "part of" as well as the "is a" hierarchy, we explicitly expose the redundancy of the system under consideration. If 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.

Also from the same class structure, there are many different ways that these objects can be instantiated and organized. No one particular architecture can really be deemed "correct." This is what makes system architecture challenging—finding the balance between the many ways the components of a system can be structured, the five attributes of complex systems, and the needs of the system user.

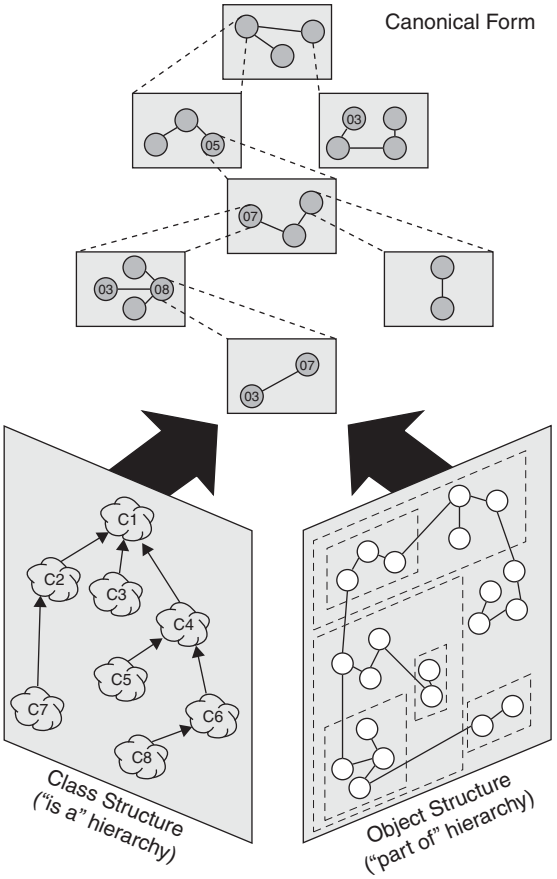



**Figure 1–2** The Canonical Form of a Complex System

Our experience is that the most successful complex software systems are those whose designs explicitly encompass well-engineered class and object structures and embody 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, that are within budget, and that meet their requirements, unless they are designed with these factors in mind.

## 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? 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 nondeterministic 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 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 on our ability to cope with this complexity. How then do we resolve this predicament?

## 1.5 Bringing Order to Chaos

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 system architects: the ones who devise innovative idioms, mechanisms, and frameworks that others can use as the architectural foundations of other applications or systems. However, “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 them” [2]. Although there is a touch of genius in all of us, in the realm of industrial-strength software we cannot always rely on divine inspiration to carry us through. Therefore, we must consider more disciplined ways to master complexity.

# The Role of Decomposition

"The technique of mastering complexity has been known since ancient times: divide et impera (divide and rule)" [16]. 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 on 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 [17].

## *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 1–3 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 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].

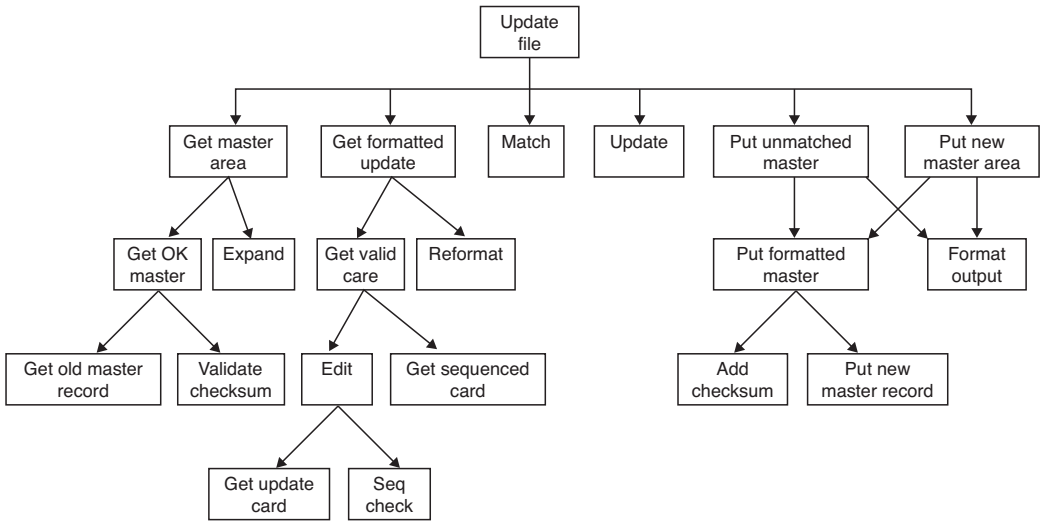



**Figure 1–3** Algorithmic Decomposition

## Object-Oriented Decomposition

We suggest that there is an alternate decomposition possible for the same problem. In Figure 1–4, 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 checksum*, we have identified objects such as *Master File* and *Checksum*, which derive directly from the vocabulary of the problem domain.

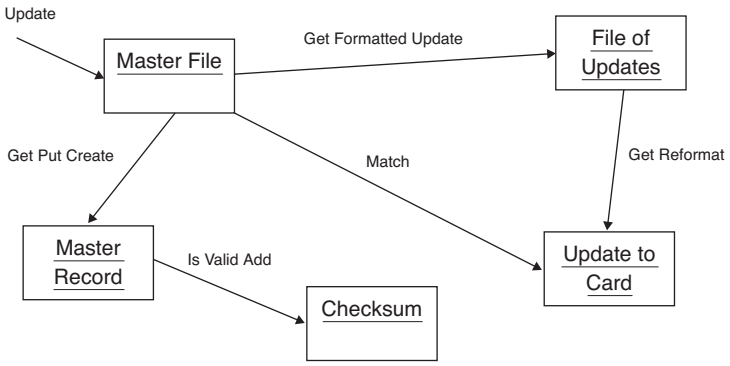



**Figure 1–4** Object-Oriented Decomposition

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 that 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 on 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 on which these operations act.

## Categories of Analysis and Design Methods

We find it useful to distinguish between the terms *method* and *methodology*. A method is a disciplined procedure for generating a set of models that describe various aspects of a software system under development, using some well-defined notation. A methodology is a collection of methods applied across the software development lifecycle and unified by process, practices, and some general, philosophical approach. Methods are important for several reasons. Foremost, they instill a discipline into the development of complex software systems. They define the products that serve as common vehicles for communication among the members of a development team. Additionally, methods define the milestones needed by management to measure progress and to manage risk.

Methods have evolved in response to the growing complexity of software systems. In the early days of computing, one simply did not write large programs because the capabilities of our machines were greatly limited. The dominant constraints in building systems were then largely due to hardware: Machines had small amounts of main memory, programs had to contend with considerable latency within secondary storage devices such as magnetic drums, and processors had cycle times measured in the hundreds of microseconds. In the 1960s and 1970s the economics of computing began to change dramatically as hardware costs plummeted and computer capabilities rose. As a result, it was more desirable and now finally economical to automate more and more applications of increasing complexity. High-order programming languages entered the scene as important tools. Such languages improved the productivity of the individual developer and of the development team as a whole, thus ironically pressuring us to create software systems of even greater complexity.

Many design methods were proposed during the 1960s and 1970s to address this growing complexity. The most influential of them was top-down structured design, also known as *composite design*. This method was directly influenced by the topology of traditional high-order programming languages, such as FORTRAN and COBOL. In these languages, the fundamental unit of decomposition is the subprogram, and the resulting program takes the shape of a tree in which subprograms perform their work by calling other subprograms. This is exactly the approach taken by top-down structured design: One applies algorithmic decomposition to break a large problem down into smaller steps.

Since the 1960s and 1970s, computers of vastly greater capabilities have evolved. The value of structured design has not changed, but as Stein observes, “Structured programming appears to fall apart when applications exceed 100,000 lines or so of code” [19]. Dozens of design methods have been proposed, many of them invented to deal with the perceived shortcomings of top-down structured design. The more interesting and successful design methods are cataloged by Peters [20], by Yau and Tsai [21], and in

a comprehensive survey by Teledyne Brown Engineering [22]. Perhaps not surprisingly, many of these methods are largely variations on a similar theme. Indeed, as Sommerville suggests, most methods can be categorized as one of three kinds [23]:

- Top-down structured design
- Data-driven design
- Object-oriented design

Top-down structured design is exemplified by the work of Yourdon and Constantine [24], Myers [25], and Page-Jones [26]. The foundations of this method derive from the work of Wirth [27, 28] and Dahl, Dijkstra, and Hoare [29]; an important variation on structured design is found in the design method of Mills, Linger, and Hevner [30]. Each of these variations applies algorithmic decomposition. More software has probably been written using these design methods than with any other. Nevertheless, structured design does not address the issues of data abstraction and information hiding, nor does it provide an adequate means of dealing with concurrency. Structured design does not scale up well for extremely complex systems, and this method is largely inappropriate for use with object-based and object-oriented programming languages.

Data-driven design is best exemplified by the early work of Jackson [31, 32] and the methods of Orr [33]. In this method, mapping system inputs to outputs derives the structure of a software system. As with structured design, data-driven design has been successfully applied to a number of complex domains, particularly information management systems, which involve direct relationships between the inputs and outputs of the system but require little concern for time-critical events.

The underlying concept of object-oriented analysis is that one should model software systems as collections of cooperating objects, treating individual objects as instances of a class within a hierarchy of classes. Object-oriented analysis and design directly reflects the topology of high-order programming languages such as Smalltalk, Object Pascal, C++, the Common Lisp Object System (CLOS), Ada, Eiffel, Python, Visual C#, and Java.

However, the fact remains that we cannot construct a complex system in both ways simultaneously, for they are completely orthogonal views.[3] We must start

3. Langdon suggests that this orthogonality has been studied since ancient times. As he states, “C. H. Waddington has noted that the duality of views can be traced back to the ancient Greeks. A passive view was proposed by Democritus, who asserted that the world was composed of matter called atoms. Democritus’ view places things at the center of focus. On the other hand, the classical spokesman for the active view is Heraclitus, who emphasized the notion of process” [34].

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 Chapter 2, 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 on 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 Applications section of this book demonstrates these benefits through several applications, drawn from a diverse set of problem domains. The sidebar in this chapter, Categories of Analysis and Design Methods, further compares and contrasts the object-oriented view with more traditional approaches to design.

## 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 on 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 (e.g., 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 [37].

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.

## 1.6 On Designing Complex Systems

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" [38]. Similarly, Dijkstra observes, "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" [39].

## Engineering as a Science and an Art

The role of the engineer as artist is particularly challenging when the task is to design an entirely new system. 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 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 on 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.

## 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 [40]

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" [41]. 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 [42]. 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 in both expected and unusual situations, and then we 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 personal computer, an electrical engineer must take into consideration the component-level view of the system as well as the physical layout of the circuit boards. This component view forms a logical picture of the design of the system, which helps the engineer to reason about the cooperative behavior of the components. The board layout represents the physical packaging of these components, constrained by the board size, available power, and the kinds of components 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 components, as well as timing diagrams that show the behavior of these components 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 Methodologies

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 methodologies, which we can loosely classify into three categories

(see the Categories of Analysis and Design Methods sidebar). Despite their differences, all of these have elements in common. Specifically, each 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 on 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 that are focused on 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 inherent in developing complex software systems.

In this chapter, we have made a case for using object-oriented analysis and design to master the complexity associated with developing software systems. Additionally, we have suggested a number of fundamental benefits to be derived from applying this method. Before we present the notation and process of object-oriented design, however, we must study the principles on which object-oriented development is founded, namely, abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence.

## Summary

- Software is inherently complex; the complexity of software systems often exceeds the human intellectual capacity.

- The task of the software development team is to engineer the illusion of simplicity.
- Complexity often takes the form of a hierarchy; it is useful to model both the "is a" and the "part of" hierarchies of a complex system.
- Complex systems generally evolve from stable intermediate forms.
- There are fundamental limiting factors of human cognition; we can address these constraints through the use of decomposition, abstraction, and hierarchy.
- Complex systems can be viewed by focusing on either things or processes; there are compelling reasons for applying object-oriented decomposition, in which we view the world as a meaningful collection of objects that collaborate to achieve some higher-level behavior.
- Object-oriented analysis and design is the method that leads us to an object-oriented decomposition; object-oriented design uses a notation and process for constructing complex software systems and offers a rich set of models with which we may reason about different aspects of the system under consideration.