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.

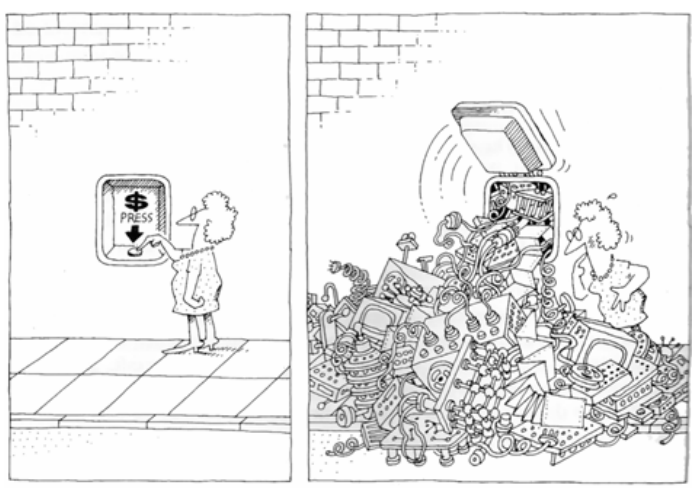
The Complexity of the Problem Domain

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.

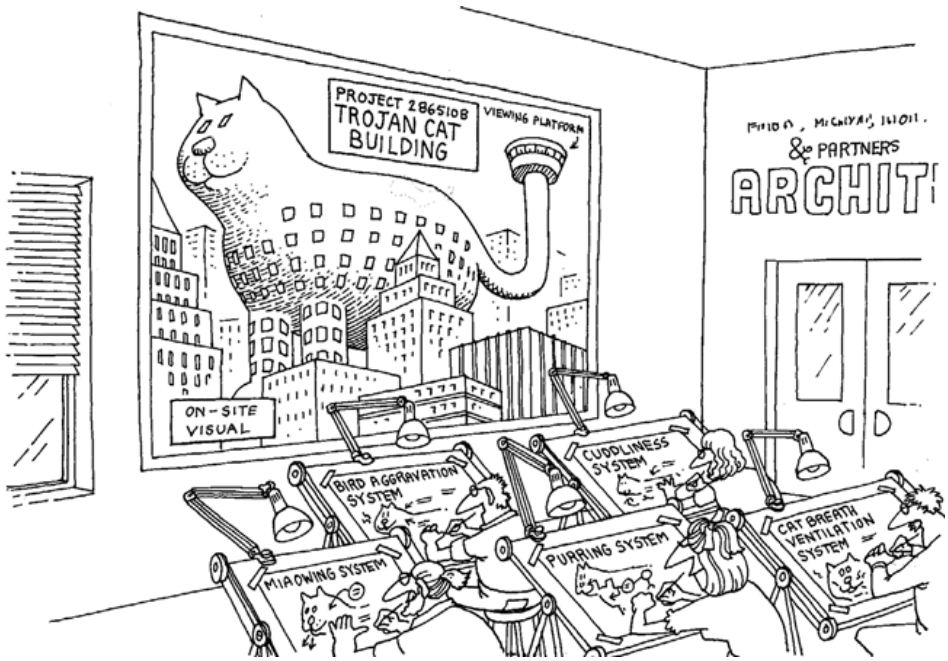
Hierarchic Structure

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.

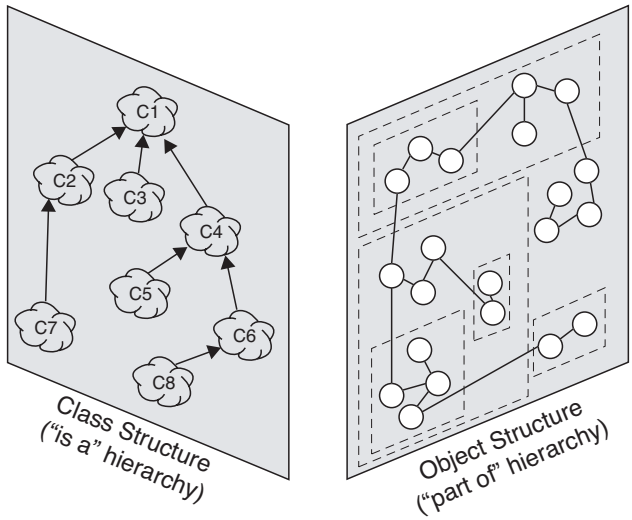


Figure 1–1 The Key Hierarchies of Complex Systems

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.

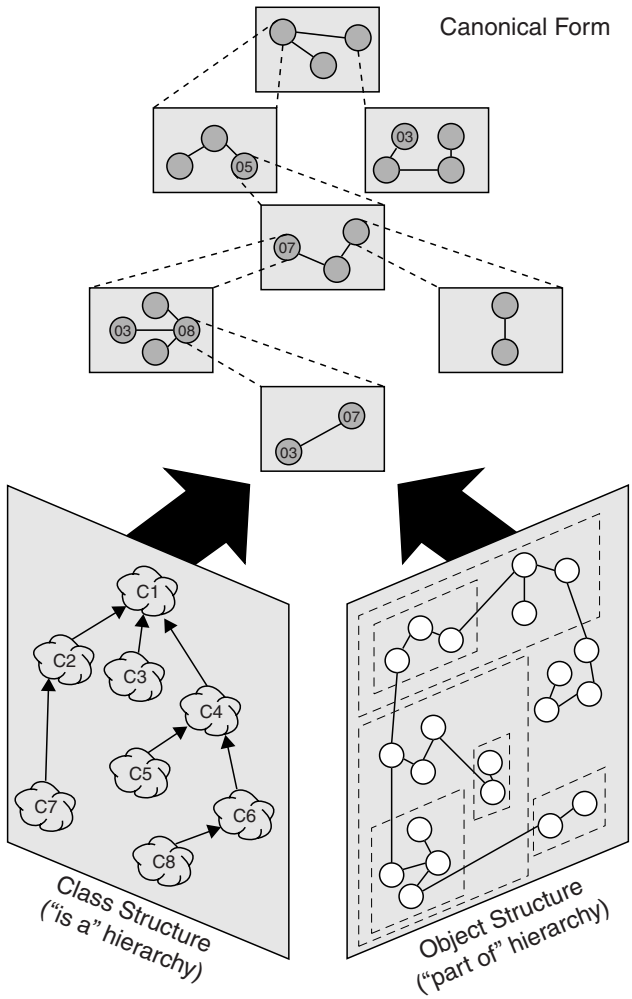


Figure 1–2 The Canonical Form of a Complex System

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.