Chapter 1: Complexity

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

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.

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*.

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.

Inherent complexity derives from four elements (Elements of Complexity) Why Software is inherently Complex ?

1. **The complexity of the problem domain** It’s a external complexity. 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.

   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 .

   Requirements Change: 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

   2. **The difficulty of managing the developmental process** :Many modules, many developers, need of proper Communication/coordination tool: No one person can ever understand 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, there are always significant challenges associated with team development. More developers mean 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.

   3. The flexibility possible through software 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 labour intensive business

   Example: 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.

4. The problems of characterizing the behaviour of discrete systems. External event may corrupt the state of a system because its designers failed to take into account certain interactions among events. If any 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. 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. In continuous system one state may not change the other state/behavior change 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. 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 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.

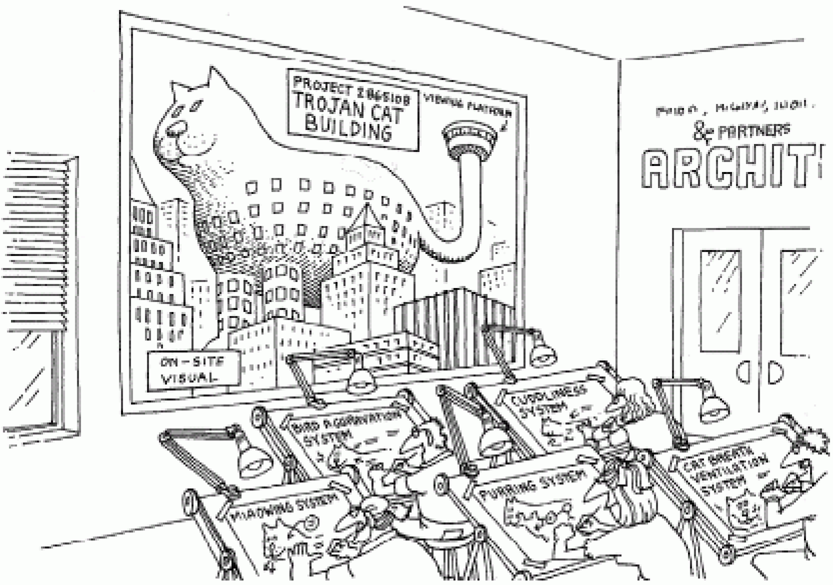
1.Hierarchic Structure

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.

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". 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".



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

2.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.

3.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 inter component linkages. This fact has the effect of separating the high-frequency dynamics of the componentsÂ—involving the internal structure of the componentsÂ—from the low-frequency dynamicsÂ—involving interaction among components.

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.

4.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.

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.

5.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". 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.

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.

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.

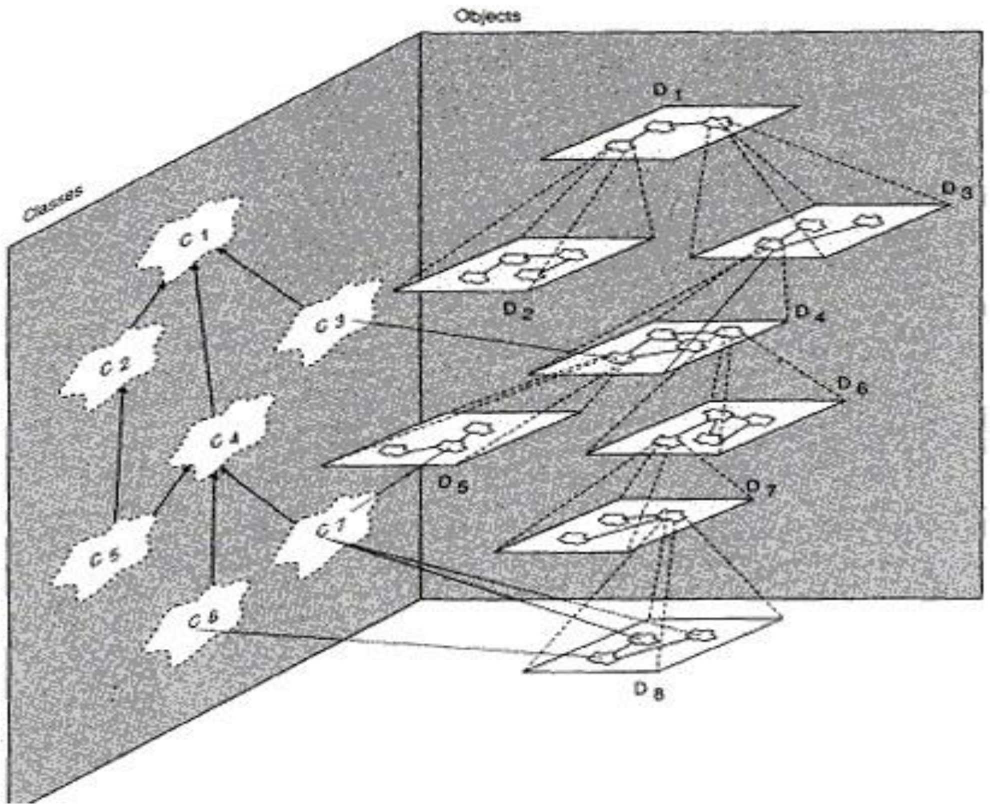


Figure 1-1

The Canonical Form of a Complex System

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.

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.

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)" [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 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 topdown 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 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

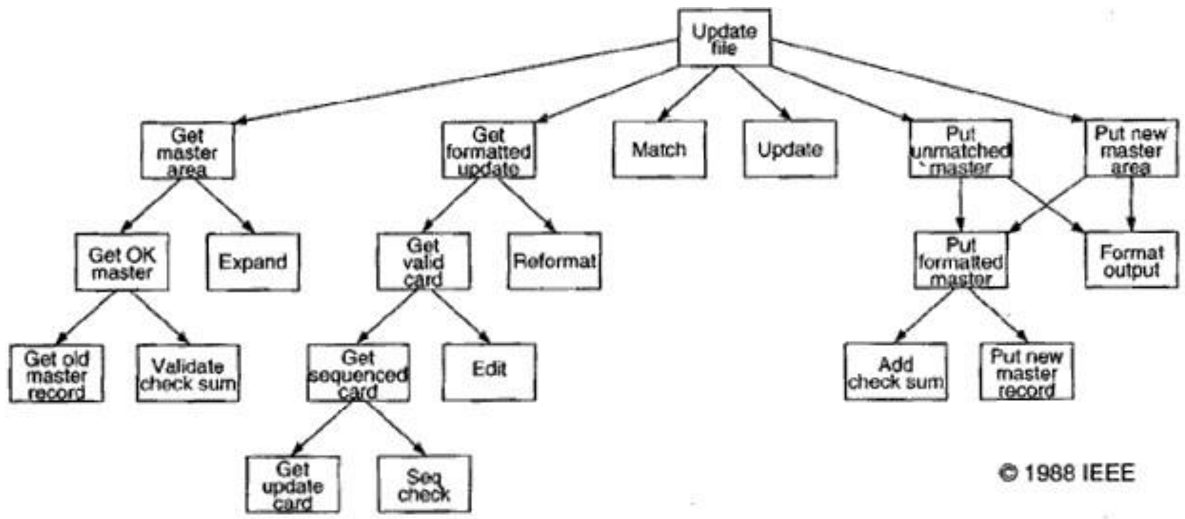


Figure 1-2

Algorithmic Decomposition

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 welldefined 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

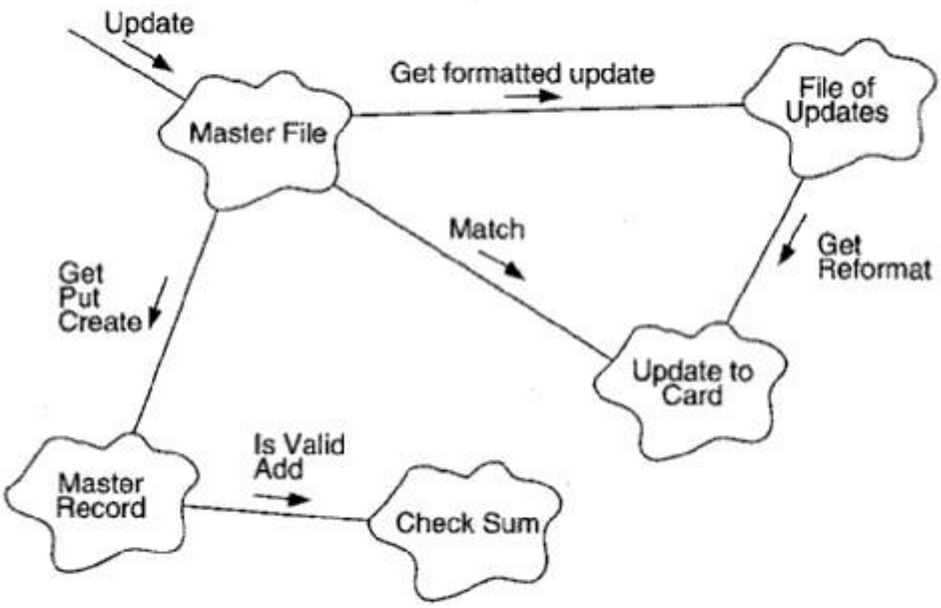

Update
Get formatted update
Master File
File of
Updates
Match
Get
Reformat
Get
Put
Create
Update to
Card
Is Valid
Add
Master
Record
Check Sum

Figure 1-3

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 views4. 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.

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" [38]. 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 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

The Elements of Software Design Methods

Clearly, there is no magic, no "silver bullet” 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:

• Process The activities leading to the orderly Notation The language for expressing each model.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

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 up 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 that are inherent in developing complex software systems

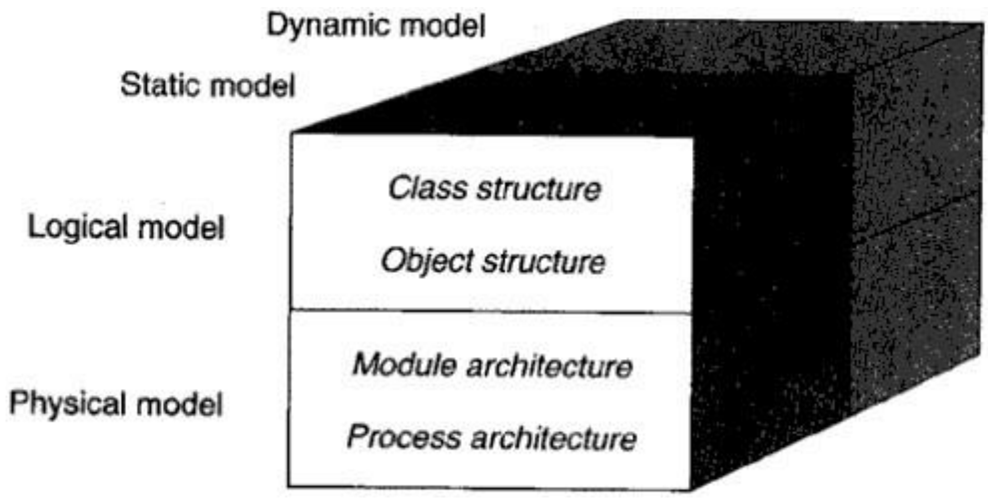


Figure 1-4 The Models of Object-Oriented Development

Evolution of the Object Model

Trends in Software Engineering

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

• The shift in focus from programming-in-the-small to programming-in-the-large

• The evolution of high-order programming languages

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

• First-Generation Languages (1954-1958)

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)**
PL/1 FORTRAN + ALGOL + COBOL
ALGOL 68 Rigorous successor to ALGOL 60

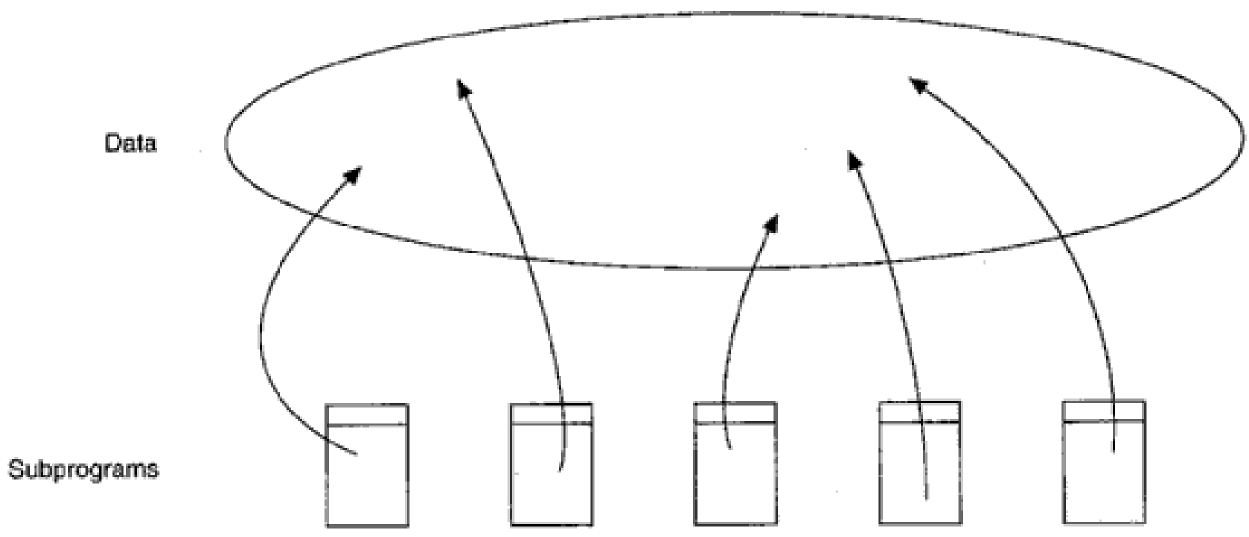
Pascal Simple successor to ALGOL 60

Simula Classes, data abstraction

• **The Generation Gap (1970-1980)**

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

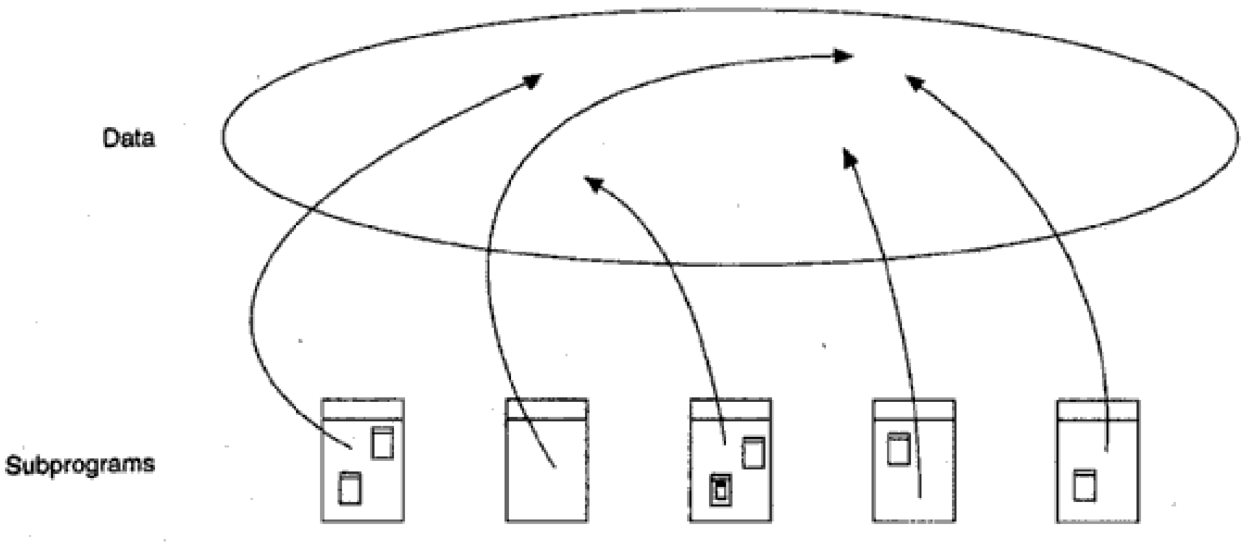
- basic physical building block of all applications is the subprogram (or paragraph)
- Example PL: FORTRAN, COBOL
- consisting only of global data (exposed to all) and subprograms
- During design, one can logically separate different kinds of data from one another, but there is no PL support that can enforce these design decisions.
- contains a tremendous amount of cross-coupling among subprograms, implied meanings of data, and twisted flows of control, thus threatening the reliability of the entire system and certainly reducing the overall clarity of the solution.



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

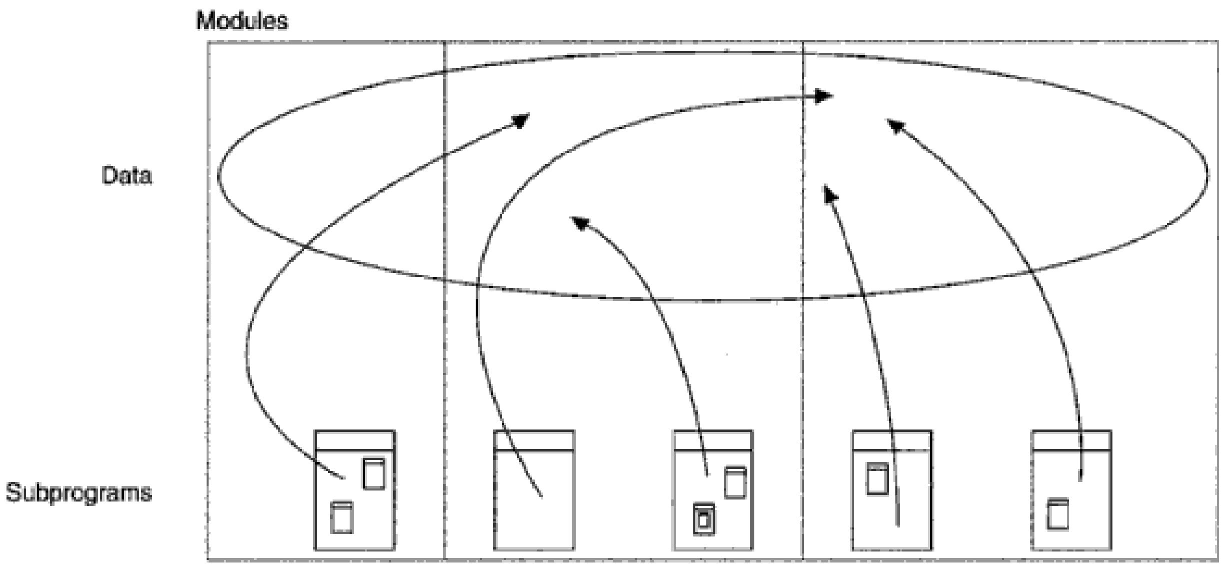
- People accepted the concept of Procedural Abstraction
    - Earlier the Sub-programs were seen simply as a labor-saving devices
    - realization that subprograms could serve as an abstraction mechanism had three important consequences
    - languages were invented that supported a variety of parameter passing mechanisms.
    - 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.
    - structured design methods emerged, offering guidance to designers trying to build large systems using subprograms as basic physical building blocks

- but it still fails to address the problems of programming-in-the-large and data design



The Topology of Late Third-Generation Programming Languages

- Another important structuring mechanism evolved to address the growing issues of programming-in-the-large.
- Larger programming projects meant larger development teams, and thus the need to develop different parts of the same program independently.answer to this need was the separately compiled module
- PL supported modular structure, but, no support to check semantic consistency among module interfaces (Prototype & Call)
- Almost no support for data abstraction and strong typing
    - errors could be detected only during execution



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.

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.

Object-Oriented Programming

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.

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.

Object-Oriented Analysis

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.

Elements of the Object Model

There are four major elements of this model:

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

There are three minor elements of the object model:

• Typing

• Concurrency

• Persistence

Abstraction

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.

Example − When a class Student is designed, the attributes enrolment_number, name, course, and address are included while characteristics like pulse_rate and size_of_shoe are eliminated, since they are irrelevant in the perspective of the educational institution

Encapsulation

Encapsulation hides the details of the implementation of an object.

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

Modularity

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

Modularity packages abstractions into discrete units.

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

The two types of hierarchies in OOA are −

- **"IS–A" hierarchy** − It defines the hierarchical relationship in inheritance, whereby from a super-class, a number of subclasses may be derived which may again have subclasses and so on. For example, if we derive a class Rose from a class Flower, we can say that a rose "is–a" flower.
- **"PART–OF" hierarchy** − It defines the hierarchical relationship in aggregation by which a class may be composed of other classes. For example, a flower is composed of sepals, petals, stamens, and carpel. It can be said that a petal is a "part–of" flower.

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 behavioural properties which a collection of entities all share" [68]. For our purposes, we will use the terms type and class interchangeably9. 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.

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

Concurrency

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 is tbe properly that distinguisbes an active object from one tbat is not active.

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

Persistence is theproperty of an object tbrougb which its existence transcends time (i.e. tbe object continues to exist after its creator ceases to exist) and/or space (i. e. the objects location moves from the address space in wbich it was created).

Applying 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.

Applications of the Object Model

The object model has proven applicable to a wide variety of problem domains. 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