UNIT-I

SOFTWARE COMPLEXITY

The Properties of Simple and Complex Software Systems

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 programmer working in isolation.

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. Learning how to design them does not interest us.

We are interested in design and development of industrial-strength software. These are the applications that exhibit a very rich set of behaviors. For example, reactive systems that drive or are driven by events in the physical world; applications that maintain the integrity of lacs of records while allowing concurrent updates and queries; and systems for the command and control of real-world entities, such as the routing of air or railway traffic.

Software systems such as these tend to have a long life span, and over time, many users come to depend upon their proper functioning.

The distinguishing characteristic of industrial-strength software is that it is intensely difficult, if not impossible, for the individual developer to comprehend all the aspects (details) of its design. The complexity of such systems exceeds the human intellectual capacity.

This complexity is an essential property of all large software systems. By essential we mean that we may master this complexity, but we can never make it go away.

Certainly, there will always be geniuses among us who can cope with such complexity. But, we cannot rely on genius people. We must consider more disciplined ways to master complexity.

Let us examine why complexity is an essential property of all software systems.

Why Software Is Inherently Complex

Brooks says: “The complexity of software is an essential property, not an accidental one"

We observe that this inherent complexity derives from four elements:

A. the complexity of the problem domain,

B. the difficulty of managing the developmental process,
C. the flexibility possible through software, and
D. the problems of characterizing the behavior of discrete systems.

A. The Complexity of the Problem Domain

The problem domain in real world involves complexity. It involves competing and contradictory requirements. For example, 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 unrestricted external (peripheral) complexity is what causes the arbitrary (random) complexity.

This external complexity arises from the mismatch between the users and developers.

The users (may have only vague ideas of what they want in a software) generally find it very hard to give precise expression to their needs in a form that developers can understand. This situation occurs because each group generally lacks expertise in the domain of the other.

Even if users had perfect knowledge of their needs, we currently have few instruments for precisely capturing these requirements.

The common way of expressing requirements today is with large volumes of text, occasionally accompanied by a few drawings. Such documents are difficult to comprehend, are open to varying interpretations.

A further complication is that the requirements of a software system often change during its development. The hands-on with prototype system makes users more clear about what they want. Similarly, during development the developers master the problem domain, enabling them to ask better questions that illuminate (clarify) the dark comers of a system's desired behavior.

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, large 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.

B. The Difficulty of Managing the Development Process

Today, it is not unusual to find delivered systems whose size is measured in millions of lines of code (in a high-order programming language). No one person can ever understand such a system completely. Even if we decompose our implementation in meaningful ways, we still end up with hundreds and sometimes thousands of separate modules.

This amount of work demands that we use a team of developers, and ideally we use as small a team as possible. However, no matter what its size, there are always significant challenges associated with team development. More developers means more complex communication and hence more difficult coordination, particularly if the team is geographically dispersed, as is often the case in very large projects.

With a team of developers, the key management challenge is always to maintain a unity and integrity of design.

C. The Flexibility Possible Through Software

A home-building company generally does not operate its own tree farm from which to harvest trees for wood. 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. It 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 (plug and play) standards exist in the software industry. As a result, software development remains a labor-intensive business.

D. The Problems of Characterizing the Behavior of Discrete Systems

If we toss a ball in to the air, we can reliably predict its path because we know that under normal conditions, certain laws of physics apply. We can predict what will happen if we threw the ball a little harder. There will not be surprising behavior from ball. But, software does not follow such behavior. The reason is that the software running on digital computer is a system with discrete states, unlike a continuous system such as the motion of the tossed ball.

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.

The system described by a continuous function 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.

We try to design our systems in such as manner (with a separation of concerns), so that the behavior in one part has minimal impact upon the behavior in another part. However, 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. To overcome such possibilities, an extensive testing of software is required. But, for all except the most trivial systems, exhaustive testing is impossible.

Since we have neither the mathematical tools nor the intellectual capacity to model the complete behavior of large discrete systems, we must be content with acceptable levels of confidence regarding their correctness.

*****The Consequences of Complexity**

The more complex the system, the more open it is to total breakdown

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.* This crisis translates into the squandering (wasting) of human resources.

Since the underlying problem springs from the inherent complexity of software, it is suggested to first study how complex systems in other disciplines are organized. We can observe successful systems of significant complexity. Some of these systems are the works of humanity, such as the Space Shuttle, the England/France tunnel, and large business organizations such as Microsoft or

General Electric. Many even more complex systems appear in nature, such as the human circulatory system or the structure of a plant.

THE STRUCTURE OF COMPLEX SYSTEMS – EXAMPLES

1. 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 storage devices.

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. We can understand 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 represents different levels of abstraction (understandable by itself)

2. The Structure of Plants and Animals

In Botany, plants are complex multi cellular 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 represents 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.

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.

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.

Turning briefly to the field of zoology, we note that multi cellular 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 soon.

The fundamental building block of all animal matter is the cell, just as the cell is the elementary structure of all plant life (with little differences).This is an example of commonality that crosses domains.

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

3. The Structure of Matter

The 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 (garbage) 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.

4. The Structure of Social Institutions

In social institutions, groups of people join together to accomplish tasks that cannot be done by individuals. 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. 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.

ATTRIBUTES OF A COMPLEX SYSTEM

There are five Attributes (properties/characteristics) common to all Complex Systems.

1. It can be Decomposed; often, in Hierarchy
2. The concept of primitive (simple) component is subjective
3. Intra-Component linkages are found stronger than Inter-Components linkages
4. It is created by combining in various ways the Limited, Simple Sub-Components
5. It evolves from simple system that worked

These five attributes are discussed below.

1. Hierarchy

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

These systems are nearly decomposable. That is, not 100% independent. Because their parts are not completely Independent.

However, this nearly decomposable hierarchy structure enable us to understand, describe and use/see such systems and its parts.

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 valued added by a system come from the relationships between the parts, not from the parts in isolation.

2. Primitive Component

The choice of what components in a system are primitive is relatively arbitrary (subjective) 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. Component Linkages

Intra-component linkages are generally stronger than intercommoning linkages. This fact has the effect of separating the high-frequency dynamics of the components - involving the internal structure of the components - from the low-frequency dynamics - involving interaction among components

4. Simple Sub Components

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. Evolve from Simple System

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 upon which more complex systems are built

THE ROLE OF DECOMPOSITION

- The technique of mastering complexity has been known since ancient times: divide and rule (process).
- 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/fulfill the constraint that exists upon the channel (focus/concentrate) capacity of human cognition (thought/reasoning/understanding): to understand any given level of a system, we need to comprehend (understand) only few parts (rather than all parts) at once.

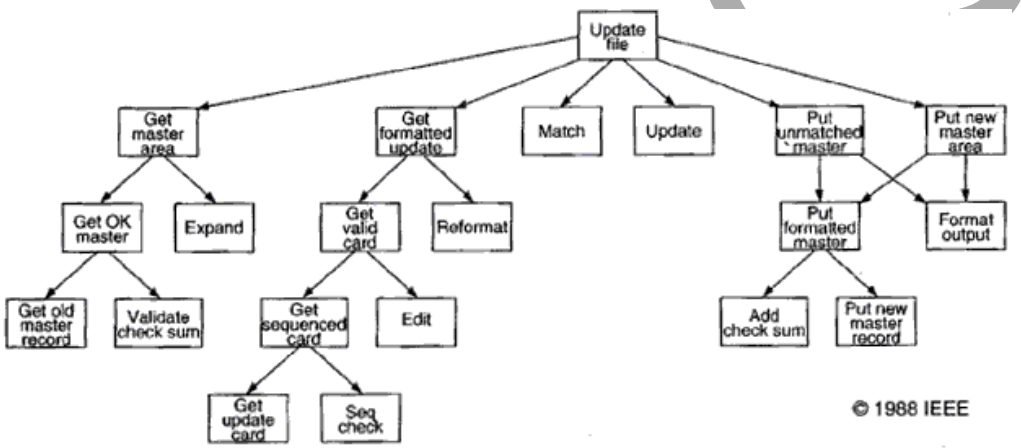
- The intelligent decomposition directly addresses the inherent complexity of software by forcing a division of a system's state space [into smaller state spaces]

There are two ways of Decomposition: Algorithmic Decomposition and Object-Oriented Decomposition.

Algorithmic Decomposition (AD)

This technique is used extensively in *Top down Structured [System Analysis &] Design* Methodology. In AD, each module in the system denotes a major step in some overall process

The following Figure shows structure chart (that shows the relationships among various functional elements of the system/solution) for part of the design of a program that updates the content of a master file



Object-Oriented Decomposition

It decomposes the system according to the *key abstractions* in the problem domain (See Following Figure).

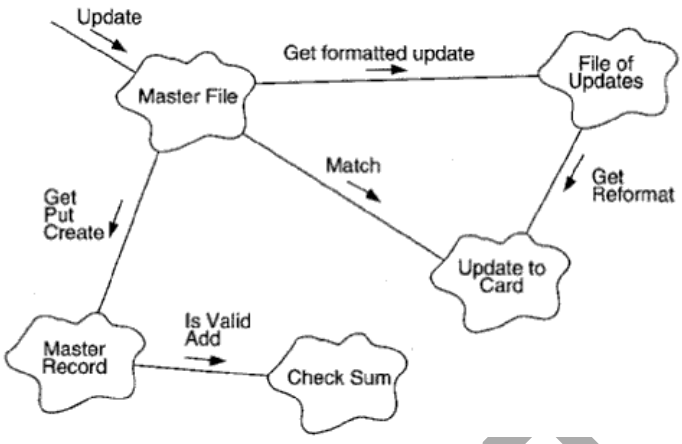
Rather than decomposing the problem into steps such as "*Get formatted update"* and "*Add check sum"*, we have identified objects such as "*Master File"* and "*Check Sum"*, which derive directly from the vocabulary of the problem domain.

In OO 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*.

Each object in our solution represents its own unique behavior, and each one model some object in the real world.

From this perspective, an object is simply a tangible (perceptible) entity which exhibits some well-defined behavior. Objects do things, and we ask them to perform what they do by sending them a message (calling a method of that object).



Algorithmic Decomposition VS. Object-Oriented Decomposition

- Although both designs solve the same problem, they do so in quite different ways
- Which is the right way to decompose a complex system - by algorithms or by objects?
    - 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, we cannot construct a complex system in both ways simultaneously (as they are orthogonal views)
- 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

It is found better to apply the object-oriented view first. It helps us to organize the inherent complexity of software system. This approach is already used by people to describe the organized complexity of complex systems as diverse as computers, plants, galaxies, and large social institutions.

The Advantages of Object Oriented Decomposition

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 (resistant) to change and thus better able to evolve over time, because their design is based upon stable intermediate forms.
- Indeed, object-oriented decomposition greatly reduces the risk of building complex software systems, because they are designed to evolve incrementally from smaller systems in which we already have confidence.
- Furthermore, object-oriented decomposition directly addresses the inherent complexity of software by helping us make intelligent decisions regarding the separation of concerns in a large state space.

THE ROLE OF ABSTRACTION

An individual can comprehend only about seven, plus or minus two, chunks (pieces) of information at one time

By organizing the division of input simultaneously into several dimensions and successively into a sequence of chunks, we manage to break this informational bottleneck. In contemporary (modern) terms, we call this process **chunking**, or **abstraction**.

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

For example, when studying how photosynthesis works in a plant, we can focus upon the chemical reactions in certain cells in a leaf, and ignore all other parts, such as the roots and stems (which are also parts of photosynthesis process otherwise).

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 objects, it is enough to study one of them, because we expect that all others will exhibit similar behavior.

For example, rather than study each individual photosynthesizing cell within a specific plant leaf, it is enough to study one such cell.

Identifying the hierarchies within a complex software system is not easy. It requires the discovery of patterns among many objects, each of which may represent some complicated behavior. But, once we identify hierarchies, it becomes easy to understand the structure of a complex a system.

CHAPTER-2

THE EVOLUTION OF THE OBJECT MODEL

The two major trends observed in SE field are

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

Most today's industrial-strength software systems are large and complex. 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 (added/perfected) 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).

Some of the more popular high-order programming languages in generations, arranged according to the language features they first introduced, are:

First-Generation Languages (1954-1958)

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

Second-Generation Languages (1959-1961)

FORTRAN-II	Subroutines, separate compilation
ALGOL 60	Block structure, data types
COBOL	Data description, file handling
Lisp	List processing, pointers, garbage collection

Third-Generation Languages (1962-1970)

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

The Generation Gap (1970-1980)

Many different languages were invented, but few endured (continued).

First-generation languages were used primarily for scientific and engineering applications, and the vocabulary of this problem domain was almost entirely mathematics, e.g., FORTRAN-I.

It represented a step closer to the problem space, and a step further away from the machine (assembly/machine language).

Among second-generation languages, the emphasis was upon algorithmic abstractions.

2G PLs included business applications also as a problem space that can be solved using computer in addition to scientific applications.

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

The advent of transistors and then integrated circuit technology reduced cost of computer hardware and at the same time increased processing capacity exponentially. Larger problems with more kind of data could now be solved. Thus, languages such as ALGOL 68 and, later, Pascal evolved with support for **data abstraction**. Now a programmer could describe the meaning of related kinds of data (their type) and let the programming language enforce these design decisions. It again moved our software a step closer to the problem domain, and further away from the underlying machine.

The 1970s provided us with a couple of thousand different programming languages and their dialects. To a large extent, the drive to write larger and

larger programs highlighted the inadequacies of earlier languages; thus, many new language mechanisms were developed to address these limitations. Few of these languages survived, however, many of the concepts that they introduced found their way into successors of earlier languages.

Thus, today we have

- Smalltalk (a revolutionary successor to Simula),
- Ada (a successor to ALGOL 68 and Pascal,with contributions from Simula, Alphard, and CLU),
- CLOS (which evolved from Lisp, LOOPS, and Flavors),
- C++ (derived from a marriage of Cand Simula), and
- Eiffel (derived from Simula and Ada).
- [Java, C#, PHP, etc.]

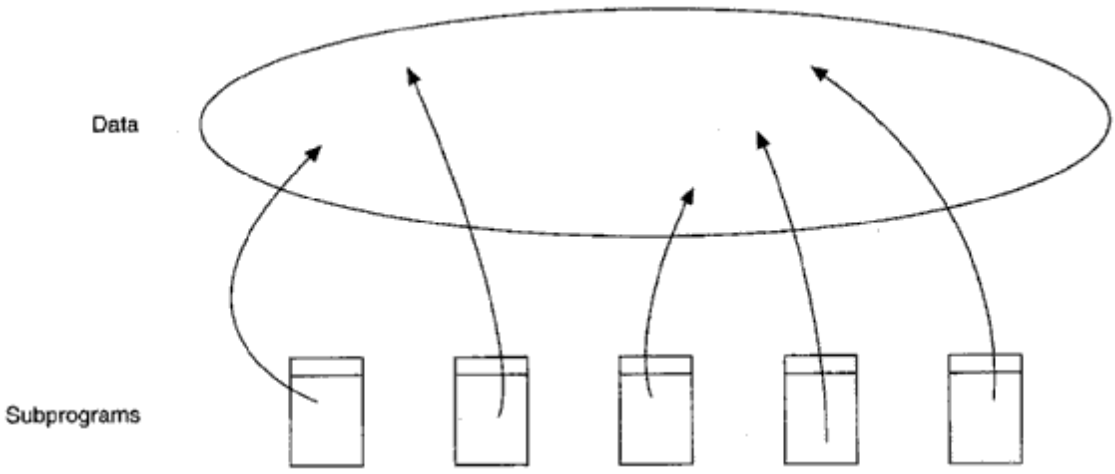
What is of the greatest interest to us is the class of languages we call **object-based and object-oriented programming languages** that best support the object-oriented decomposition of software.

THE PROGRAMMING LANGUAGE (PL) TOPOLOGY

The PL topology means the basic physical building blocks of the language and how those parts can be connected.

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

The following Figure shows the topology of most first-and early second-generation programming languages.



In this figure, we see that for languages such as FORTRAN and COBOL, the basic physical building block of all applications is the subprogram (or the paragraph, for those who speak COBOL).

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

The arrows in this figure indicate dependencies of the subprograms on various data. During design, one can logically separate different kinds of data from one another, but there is little in these languages that can enforce these design decisions.

An error in one part of a program can have a shocking negative effect across the rest of the system, because the global data structures are exposed for all subprograms to see.

When modifications are made to a large system, it is difficult to maintain the integrity of the original design.

A program written in these languages contains a tremendous amount of cross-coupling among subprograms, implied meanings of data, and twisted flows of control, thus threatening the reliability of the entire system and certainly reducing the overall clarity of the solution.

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

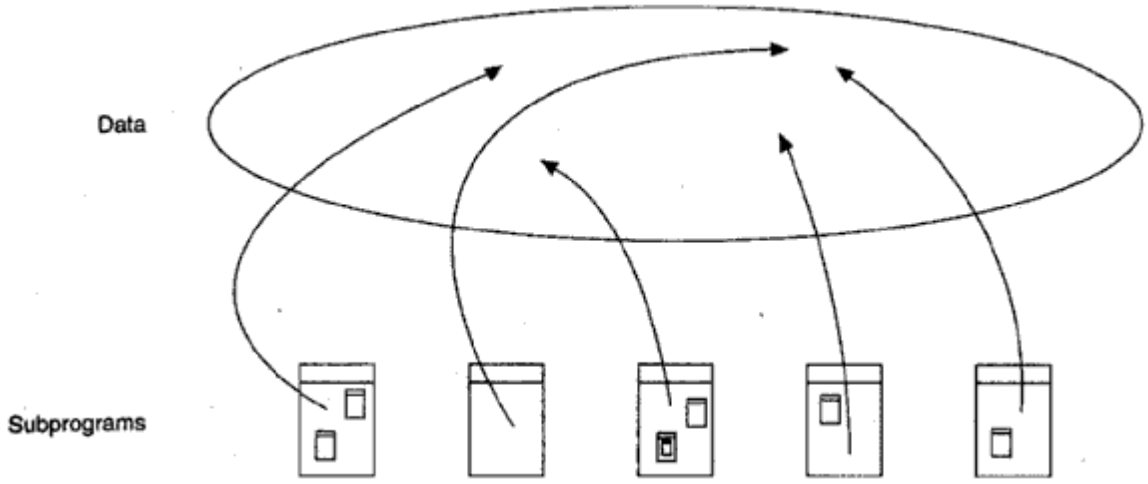
By the mid-1960s, programs were recognized as important intermediate points between the problem and the computer. The first software abstraction, now called the ***procedural abstraction***, grew directly out of this belief. Subprograms were invented prior to 1950, but were not fully appreciated as abstractions at the time. Instead, they were originally seen as labor-saving devices.

The realization that subprograms could serve as an abstraction mechanism had three important consequences.

1. First, languages were invented that supported a variety of parameter passing mechanisms.
2. Second, the foundations of structured programming were laid, manifesting themselves in language support for the nesting of subprograms and the development of theories regarding control structures and the scope and visibility of declarations.
3. Third, structured design methods emerged, offering guidance to designers trying to build large systems using subprograms as basic physical building blocks.

The following figure shows topology of these PLs. This topology addresses some of the inadequacies of earlier languages, namely, the need to have greater

control over algorithmic abstractions, but it still fails to address the problems of programming-in-the-large and data design.

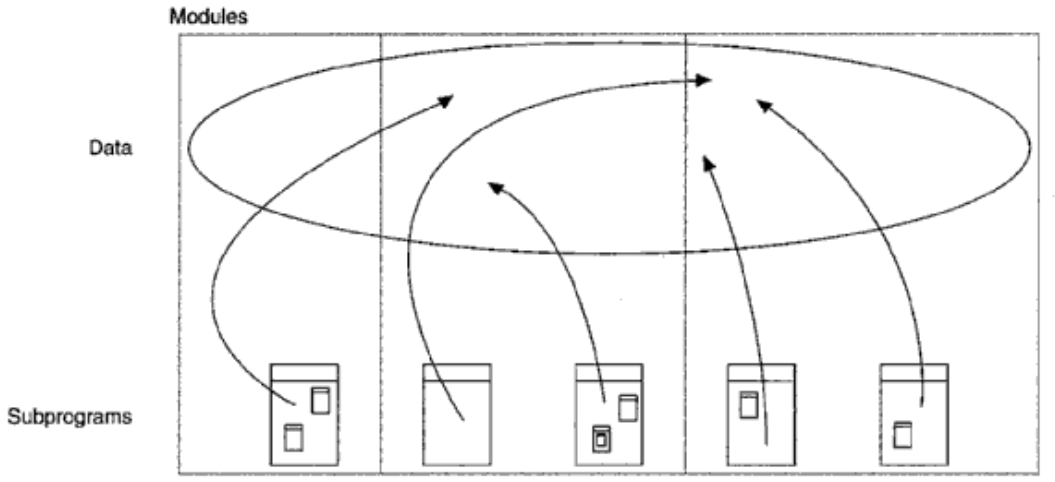


The Topology of Late Third-Generation Programming Languages

Starting with FORTRAN II, and appearing in most late third-generation program languages, another important structuring mechanism evolved to address the growing issues of programming-in-the-large.

Larger programming projects meant larger development teams, and thus the need to develop different parts of the same program independently.

The answer to this need was the separately compiled module (see following Figure). Modules were rarely recognized as an important abstraction mechanism; in practice they were used simply to group subprograms that were most likely to change together.



Most languages of this generation, while supporting some sort of modular structure, had few rules that required semantic consistency among module interfaces. A developer writing a subprogram for one module might assume that it would be called with three different parameters: a floating-point number, an array of ten elements, and an integer representing a Boolean flag. In another module, a call to this subprogram might incorrectly use actual parameters that: violated these assumptions: an integer, an array of five elements, and a negative number.

Similarly, one module might use a block of common data which it assumed as its own, and another module might violate these assumptions by directly manipulating this data.

Unfortunately, because most of these languages had dismal (low) support for data abstraction and strong typing, such errors could be detected only during execution of the program.

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

The importance of data abstraction to mastering complexity can be stated as under:

"The nature of *abstractions* that may be *achieved through the use of* ***procedures*** *is well suited to the description of* ***abstract operations***, *but is not particularly well suited to the description of* ***abstract objects***."

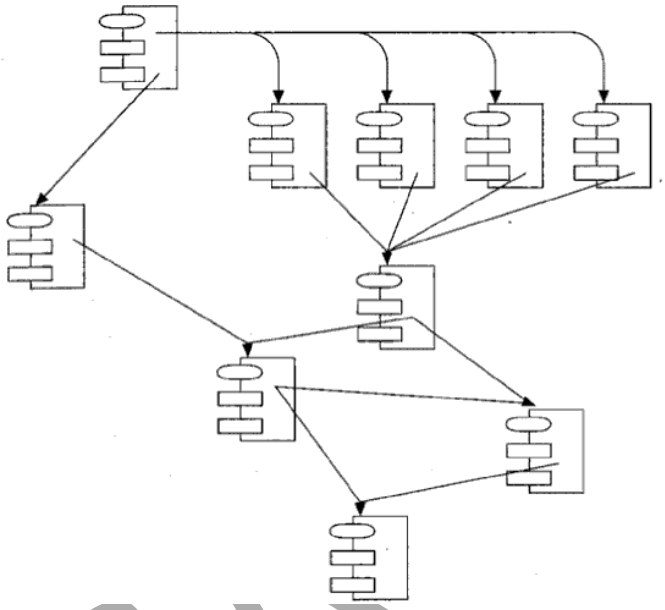
This is a serious drawback. The complexity of the data objects to be manipulated contributes substantially to the overall complexity of the problem

This realization had two important consequences.

1. First, data-driven design methods emerged, which provided a disciplined approach to the problems of doing data abstraction in algorithmically oriented languages.
2. Second, theories regarding the concept of a type appeared which eventually found their realization in languages such as Pascal.

These ideas first appeared in the language Simula and were improved upon in several languages such as Smalltalk, Object Pascal, C++, CLOS, Ada, and Eiffel. These languages are called *object-based or object-oriented*.

The following Figure illustrates the topology of these languages for small to moderate sized applications. The physical building block in these languages is the *module*, which represents a logical collection of classes and objects instead of subprograms, as in earlier languages.

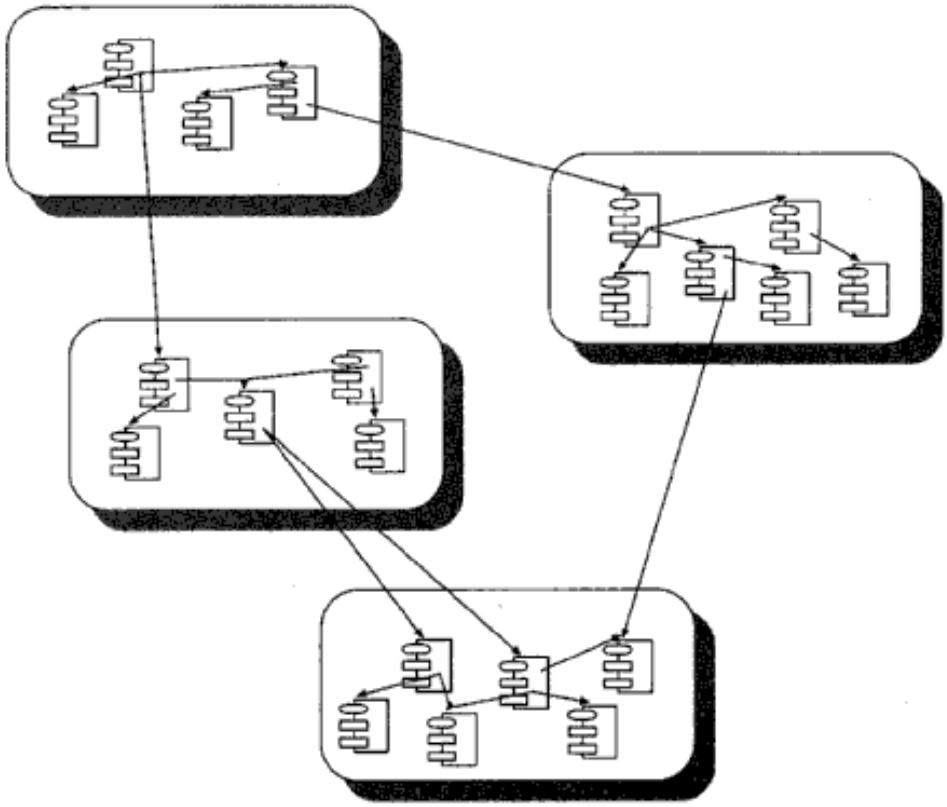


To state it another way, "lf procedures and functions are verbs and pieces of data are nouns, a procedure-oriented program is organized around verbs while an object-oriented program is organized around nouns".

For this reason, the physical structure of a small to moderate-sized object-oriented application appears as a graph, not as a tree, which is typical of algorithmically oriented languages. Additionally, there is little or no global data. Instead, data and operations are united in such a way that the fundamental logical building blocks of our systems are no longer algorithms, but instead are classes and objects.

By now we have progressed beyond programming-in-the-large and must cope with programming-in-the-colossal (huge). That is, demand for further larger and more complex systems is encountered in real world.

For very complex systems, we find that classes, objects, and modules provide an essential yet insufficient means of abstraction. Fortunately, the object model scales up. In large systems, we find clusters of abstractions built in layers on top of one another. At any given level of abstraction, we find meaningful collections of objects that collaborate to achieve some higher-level behavior. If we look inside any given cluster to view its implementation, we uncover yet another set of cooperative abstractions. This is exactly the organization of complexity (described earlier); this topology is shown in following Figure.



----------******* UNIT-I Over *********----------**