Chapter1: Complexity

As Brooks suggests, "The complexity of software is an essential property, not an accidental one“

Software is inherently complex; the complexity of software systems often exceeds the human intellectual capacity. The task of the software development team is to engineer the illusion of simplicity

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 size, 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 labor-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 on-site 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 behavior 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.

Common Attributes of Complex System

1. **Hierarchical and interacting subsystems**

   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.

2. **Arbitrary determination of primitive components**

   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.
3. **Stronger intra-component than inter-component link**

   Intra-component linkages are generally stronger than inter-component 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. **Combine and arrange common rearranging subsystems**

   Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements . i.e complex systems have common patterns.
5. **Evolution from simple to complex systems**

   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

Managing the Complexity

There are mainly three things which plays the vital role for managing the complexity, which are:

1. Decomposition
2. Abstraction
3. Hierarchical

1. The Role of Decomposition

The technique of mastering complexity has been known since ancient times: *divide et impera* (divide and rule)". When designing a complex software system, it is essential to decompose it into smaller and smaller parts, each of which we may then refine independently. to understand any given level of a system, we need only comprehend a few parts (rather than all parts) at once. Intelligent decomposition directly addresses the inherent complexity of software by forcing a division of a system's state space.

Decomposition Types:

1. Algorithmic Decomposition
2. Object-Oriented Decomposition

Algorithmic Decomposition

- Each module in the system denotes a major step in some overall process
- Top –down structured design ,so this approaches of decomposition is a simple approach of decomposition
- The decomposition structure chart shows the relationships among various functional elements of the solution
- Fig below is the particular structure chart, which illustrates part of the design of a program that updates the content of a master file

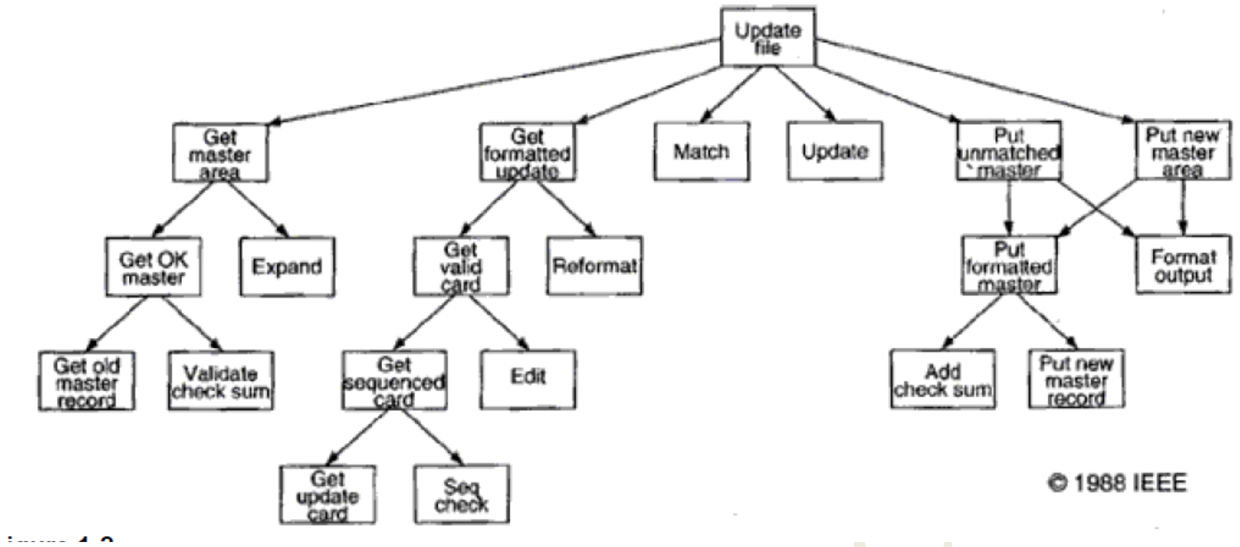
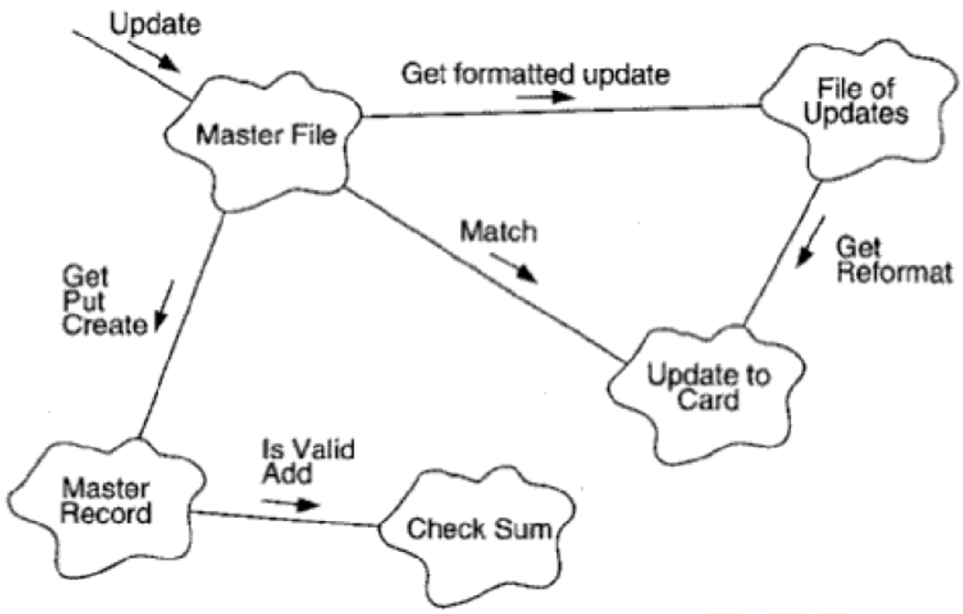


Fig: Algorithmic Decomposition

Object Oriented Decomposition

- Decomposes the system according to the key abstractions in the problem domain, rather than decomposing the problem into steps
- Bottom up approach of decomposition
- In Object oriented decomposition we view the world as a set of autonomous agents that collaborate to perform some higher level behavior.
- In this decomposition method ,each object in our solution embodies its own behavior and each one models some object in the real world.
- E.g. Here, rather than decomposing the problem into steps such as *Get Formatted Update* and *Add Checksum* we have identified objects such as *Master File and Checksum* ,which derive directly from the vocabulary of problem domain. Although both designs solve the same problem they do so in quite different way.



Algorithmic Vs Object Oriented Decomposition

1. 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 operation acts
2. Algorithmic decomposition is a top down approach where as Object oriented decomposition is bottom up approach of decomposition
3. Object oriented approach is better at helping us organizing the inherent complexity of software systems , just as it helped us to describe the organized complexity of complex systems
4. Object Oriented decomposition yields smaller systems through the reuse of common mechanism thus it provides an important economy of expressions
5. OO Systems are also more resilient to change and thus better able to evolve over time
6. OOD 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
7. OO decomposition directly addresses the inherent complexity of software by helping us make intelligent decisions regarding the separation of concerns in a large state space.

Role of Abstraction

Abstraction is powerful technique for dealing with complexity. If we 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.

Abstraction is the selective examination of certain aspects of a problem while ignoring the remaining aspects of the problem. This mechanism allows us to represents a problem in a simpler way by omitting unimportant details.

An abstractions means that each object hides from other objects the exact way in which its internal information is organized and manipulated. It only provides a set of methods which other objects can use for accessing and manipulating this private information 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

Role of Hierarchy

Explicitly recognizing the class and object hierarchies within a complex software system is another way to increase the semantic content of individual chunks of information is i.e . while designing a complex software system ,by finding hierarchies of objects and their corresponding classes we can manage complexity easily.

Object structure illustrates how different objects collaborate with one another through patterns of interaction that we call mechanisms.

The class structure highlights common structure and behavior within a system.

By classifying objects into groups of related abstractions (for example, kinds of plant cells versus animal cells), we come to explicitly distinguish the common and distinct properties of different objects, which further helps us to master their inherent complexity.

Identifying the hierarchies within a complex software system is often not easy, because it requires the discovery of patterns among many objects, each of which may embody some tremendously complicated behavior. Once we have exposed these hierarchies, however, the structure of a complex system, and in turn our understanding of it, becomes vastly simplified.

The Meaning of Design

Design encompasses the disciplined approach we use to invent a solution for some problem, so design providing a path from requirements to implementation. In terms of Software Engineering design is the process of defining and developing Architecture, data structures, algorithm and selecting proper database architecture and data types.

Purpose of design

The purpose of the design is to construct a system that:

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

As Stroustrup suggests, "the purpose of design is to create a clean and relatively simple internal structure, sometimes also called architecture. A design is the end product of the design process".

Design involves balancing a set of competing requirements. The products of design are models that enable us to reason about our structures, make trade-offs when requirements conflict, and in general, provide a blueprint for implementation.

The Elements of Software Design Methods

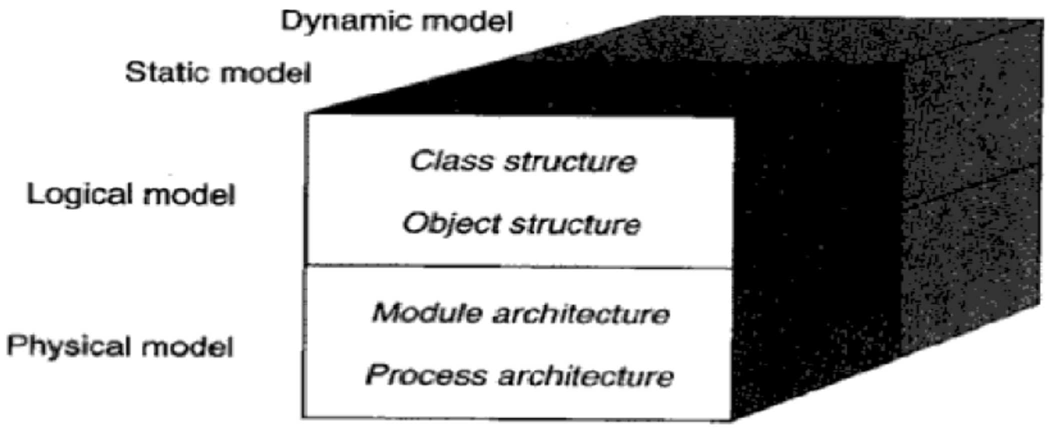
There is no magic, no "silver bullet" , that: can unfailingly lead the software engineer down the path from requirements to the implementation of a complex software system

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

The Models of Object-Oriented Development

Building models is a best way to decompose a complex system into manageable and simple system structure.

Model building is so important to the systems, object-oriented development offers a rich set of models which are used to describe complex system . The different types of models are shown in fig below.



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

Importance of Model Building

Model building appeals to the principles of decomposition, abstraction, and hierarchy. Each model within a design describes a specific aspect of the system under consideration. As much as possible, we seek to build new models upon old models in which we already have confidence.

Models give us the opportunity to fail under controlled conditions. We evaluate each model under both expected and unusual situations, and then alter them when they fail to behave as we expect or desire.
In order to express all the subtleties of a complex system, we must use more than one kind of model. For example, when designing a single-board computer, an electrical engineer must take into consideration the gate-level view of the system as well as the physical layout of integrated circuits on the board. **Gate level view: logical model, physical layout: physical view**