UNIT 1

Introduction: The Structure of Complex systems, The Inherent Complexity of Software, Attributes of Complex System, Organized and Disorganized Complexity, Bringing Order to Chaos, Designing Complex Systems, Evolution of Object Model, Foundation of Object Model, Elements of Object Model, Applying the Object Model.

1.1 The Structure of Complex systems

A complex system is a system composed of many components which may interact with each other. In many cases it is useful to represent such a system as a network where the nodes represent the components and the links their interactions. Examples of complex systems are Personal Computer, Plants, and Education System.

The structure of personal computer

A personal computer is a device of moderate complexity. Most are composed of the same major elements: a central processing unit (CPU), a monitor, a keyboard, and some sort of secondary storage device, usually either a CD or DVD drive and hard disk drive. We may take any one of these parts and further decompose it. For example, a CPU typically encompasses primary memory, an arithmetic/logic unit (ALU), and a bus to which peripheral devices are attached. Each of these parts may in turn be further decomposed: An ALU may be divided into registers and random control logic, which themselves are constructed from even more primitive elements, such as NAND gates, inverters, and so on.

Central Processing Unit (CPU) — *Executes programs*

- Arithmetic & Logic Unit (ALU)
  - Registers
    - NAND gate
      - CMOS Gate
      - Interconnect
    - Inverter
  - Control Logic
- Primary Memory
- Bus — *peripheral devices*

- Hard disk drive — *Persistent storage for data*
- Monitor — *Outputs data*
- Keyboard — *Inputs data*
- Secondary storage device (DVD drive / USB) — *Removable storage for data*

Complex systems hierarchies

- Each Level of Hierarchy represents a Layer of Abstraction
- Each Layer
  - Is built on top of other layers: *CMOS Gates —> NAND Gates—> Registers*
  - (in turn) Supports other layers: *CMOS Gates —> NAND Gates—> Registers*

        - Is independently understandable: *CPU*
        - Works independently with clear separation of concerns: *ALU, Memory*
    - Common services / properties are shared across Layers: *Same power-tree feeds the components of CPU*
    - Layers together show **Emergent Behavior**
    - Behavior of the whole is greater than the sum of its parts
    - Systems demonstrate cross-domain commonality: *Cars have processors, memory, display*

1.2 The Inherent Complexity of Software

There are two types of software are there in terms of complexity

- Simple Software
- Industrial-strength software

Simple Software

Simple Software is software with limited set of behaviors and not very complex. It is specified, constructed, maintained, and used by the same person or a small group usually the amateur programmer or the professional developer working in isolation. Such systems tend to have a very limited purpose and a very short life span. Can be thrown away and replaced with entirely new software rather than attempt to reuse them, repair them, or extend their functionality

Industrial-strength software

Industrial-strength software applications exhibits a very rich set of behaviors, 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. Complexity of Industrial-Strength Software systems exceeds the human intellectual capacity

Why Software Is Inherently Complex?

Inherent complexity derives from four elements:

1. The complexity of the problem domain,
2. The difficulty of managing the development process,
3. The flexibility possible through software,
4. The problems of characterizing the behavior of discrete systems.

The complexity of the problem domain

Domains are difficult to understand. Consider the requirements for the electronic system of a multiengine aircraft, a cellular phone switching system, or an autonomous robot. The raw functionality of such systems is difficult enough to comprehend. The functional requirements

complex to master and often are competing, even contradictory. Non-functional Requirements (usability, performance, cost, survivability, and reliability) are often Implicit and difficult to justify in the budget.
This external complexity usually springs from the ―communication gap‖ that exists between the users of a system and its developers. Users cannot express; developers cannot understand due to lack of expertise across domains.

The difficulty of managing the development process
The task of the software development team is to engineer the illusion of simplicity. We strive to write less code by inventing clever and powerful mechanisms. Today, it is not unusual to find delivered systems whose size is measured in hundreds of thousands or even millions of lines of code. The complexity of software goes well beyond the comprehension of a single (or small group of) individual/s, even with meaningful decomposition. Hence, we need more developers and more teams. The key management challenge is to maintain a unity and integrity of management.

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 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.

Problems of characterizing the behavior of discrete systems
Software is built and executed on digital computers. Hence, they are Discrete Systems. A large application would have hundreds or even thousands of variables and more than one (often several) thread of control. A state is entire collection of variables, the current values of variables, the current address of each process, the calling stack of each process. Software has finite number of states. Yet, many of these the states are often intractable and influenced by external factors. This change is not deterministic (unlike continuous system) and needs extensive Testing.

1.3 Attributes of Complex System

1. Hierarchic Structure
2. Relative Primitives
3. Separation of Concerns
4. Common Patterns
5. Stable Intermediate Forms

Hierarchic Structure
- All **systems** are composed of interrelated **sub-systems**
- Sub-systems are composed of sub-sub-systems, and so on

- Lowest level sub-systems are composed of **elementary components**
- 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
- **We can understand only those systems that have a hierarchic structure**

Relative Primitives

- **Subjective Choice** — strongly dependent on the experience and expertise of the designer
- What is primitive for one observer may be at a much higher level of abstraction for another.
- **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**

Separation of Concerns

- Hierarchic systems are:
    - **decomposable** — *can be divided into identifiable parts*
    - **nearly decomposable** — *the parts are not completely independent*
- Difference between intra- and intercomponent interactions provides a clear Separation of Concerns among the various parts of a system — helps the analysis and design in isolation

Common Patterns

- Complex systems have **Common Patterns**
- Complex systems are composed of only a few different kinds of subsystems in various combinations and arrangements (cells found in both plants and animals etc.)
- Common Patterns are a major source of reuse in OOAD. Examples include Design Patterns, STL in C++, Data Structures in Python etc.

Stable Intermediate Forms

- It is extremely difficult to design a complex system correctly in one go
- Start with a simple system and then refine (Iterative Refinement)
- Objects, once considered complex, become the primitive objects on which more complex systems are built
- The system matures from one intermediate form to the next
- **A complex system that works is invariably found to have evolved from a simple system that worked**

1.4 Organized and Disorganized Complexity

The Canonical Form of a Complex System: The discovery of common abstractions and mechanisms greatly facilitates our understanding of complex system. Because we can apply knowledge of (gained from) one system to other system (e.g., If you can drive one type of car, it‘s easier for you to drive another type of car)

Hierarchies are of multiple types

- Decomposition — part—of or HAS—A
- Abstraction — IS—A

It is essential to view system from both perspectives. ―Part of‖ hierarchy is also known as Class Structure. ―IS A‖ hierarchy is also known as Object Structure. For example an aircraft may be studied by decomposing it into its propulsion system. Flight control system and so on the decomposition represent a structural or "part of" hierarchy.

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. 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.

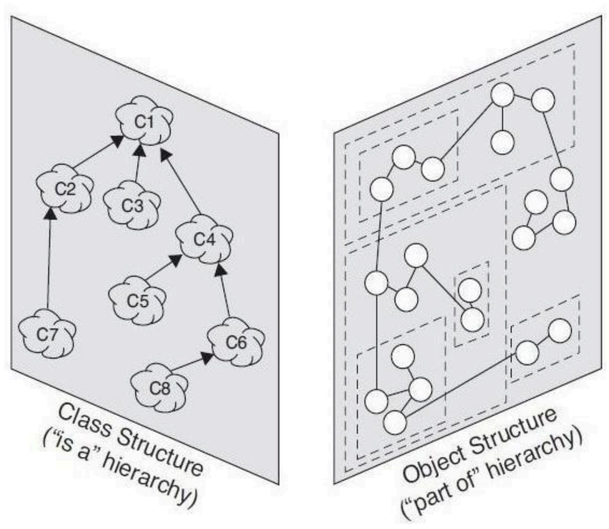


Figure 1–1 The Key Hierarchies of Complex Systems

The Limitations of the Human Capacity for Dealing with Complexity

Maximum number of chunks of information that an individual can simultaneously comprehend is on the order of seven, plus or minus two. Human channel capacity is related to the capacity of short-term memory. The Processing speed is a limiting factor — it takes the mind about five seconds to accept a new chunk of information. This leads to **Disorganized Complexity**

1.5 Bringing Order to Chaos

Principles that will provide basis for development

- Decomposition
- Abstraction
- Hierarchy

The Role of Decomposition
Decomposition techniques are divided into two types they are

- Algorithmic Decomposition
- Object-Oriented Decomposition

Algorithmic Decomposition: Decomposition is important techniques for copying with complexity based on the idea of divide and conquer. In dividing a problem into a sub problem the problem becomes less complex and easier to overlook and to deal with. Repeatedly dividing a problem will eventually lead to sub problems that are small enough so that they can be conquered. After all the sub problems have been conquered and solutions to them have been found, the solutions need to be composed in order to obtain the solution of the whole problem. The history of computing has seen two forms of decomposition, process-oriented (Algorithmic) and object-oriented decomposition.

Object-Oriented Decomposition: Objects are identified as Master file and check sum which derive directly from the vocabulary of the problem. We view the world as a set of autonomous agents that collaborate to perform some higher level behavior. Calling one operation creates another object. In this manner, each object embodies its own unique behavior. Each hierarchy in layered with the more abstract classes and objects built upon more primitive ones especially among the parts of the object structure, object in the real world. Here decomposition is based on objects and not algorithms.

The Role of Abstraction: Abstraction is an exceptionally powerful technique for dealing with complexity. Unable to master the entirely 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 about 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 roots and stems. Objects are abstractions of entities in the real world.
In general abstraction assists people's understanding by grouping, generalizing and chunking information.
Object-orientation attempts to deploy abstraction. The common properties of similar objects are defined in an abstract way in terms of a class. Properties that different classes have in common are identified in more abstract classes and then an _is-a' relationship defines the inheritance between these classes.

The Role of Hierarchy: Increase the semantic content of individual chunks of information by explicitly recognizing the class and object hierarchies. Object structure illustrates how different objects collaborate with one another through patterns of interaction that we call mechanisms Class structure highlights common structure and behavior within a system.

1.6 Designing Complex Systems

Engineering as a Science and an Art: Every engineering discipline involves elements of both science and art. The programming challenge is a large scale exercise in applied abstraction and thus requires the abilities of the formal mathematician blended with the attribute of the competent engineer. The role of the engineer as artist is particularly challenging when the task is to design an entirely new system.
The meaning of Design: 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 available for doing the design.

The Importance of Model Building: The buildings of models have a broad acceptance among all engineering disciplines largely because 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. Models give us the opportunity to fail under controlled conditions. We evaluate each model under both expected and unusual situations and then after them when they fail to behave as we expect or desire. More than one kind of model is used on order to express all the subtleties of a complex system.

The Elements of Software design Methods: Design of complex software system involves an incremental and iterative process. Each method includes the following:

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

The models of Object Oriented Development: 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 over 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.

1.7 Evolution of Object Model

The elements of the object oriented technology collectively known as the object model. The object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency and persistency. The object model brought together these elements in a synergistic way.

Generations of Programming Languages

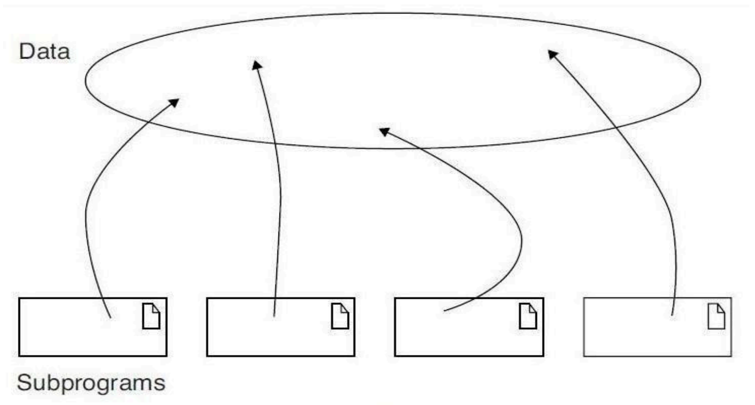
- First-generation Languages (1954-1958)
- Second-generation Languages (1959-1961)
- Third-generation Languages (1962-1970)
- Fourth-generation Languages (1970-1980)
- Object-orientation Boom (1980-1990)
- Emergence of Frameworks (1990—today)

First-generation Languages (1954-1958)

- *Major Features*
    - **Formula Translation**
    - **Mathematical Computation**
    - **Notions of Programming:**
        - Variable and Literal
        - Expressions
        - Unconditional & Conditional Flow
- *Languages*
    - FORTRAN I (Mathematical expressions)
    - ALGOL 58 (Mathematical expressions)
    - Flowmatic (Mathematical expressions)
    - IPL V (Mathematical expressions)

Topology of First-generation Languages

- Applications in these languages exhibit:
    - Flat physical structure
    - Only of global data and subprograms
    - Error in one part of a program can have effect across the rest of the system



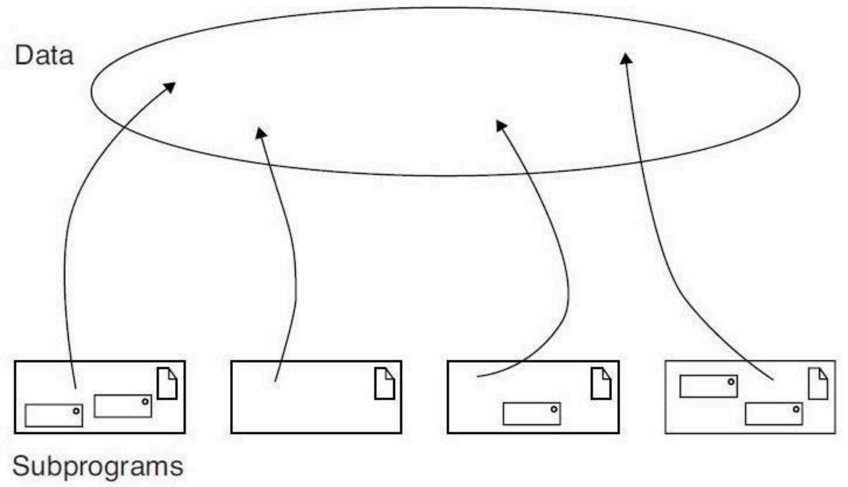
The Topology of First- and Early Second-Generation Programming Languages

Second-generation Languages (1959-1961)

- *Major Features*
    - **Blocks and Subprograms**
    - **Notions of Programming:**
        - Data Type & Description
        - Statements e Parameter Passing Mechanisms
        - Files
        - Pointers
        - List
        - Data Structures
- *Languages*
    - FORTRAN II (Subroutines, separate compilation)
    - ALGOL 60 (Block structure, data types)
    - COBOL (Data description, file handling)
    - Lisp (List processing, pointers, garbage collection)

Topology of Second-generation Languages

- Applications in these languages exhibit:
    - Foundations of structured programming - nested subprograms, control structures and scope and visibility of declarations
    - Support parameter-passing mechanisms
    - Fails to address the problems of programming-in-the-large and data design



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

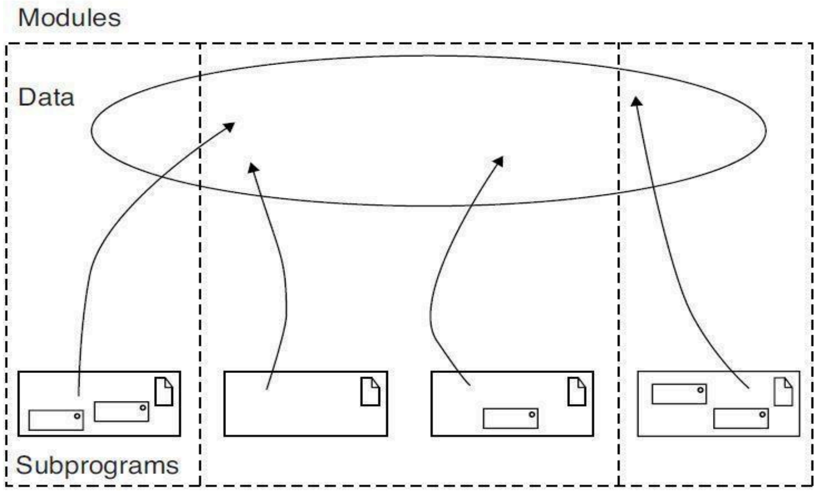
Third-generation Languages (1962-1970)

- *Major Features*
    - **Structured Programming**
    - **Modularity**

        - **Abstraction**
        - **Notions of Programming:**
            - Control Constructs
            - Class
        - **One-language-fits-it-all**
    - *Languages*
        - PL/1 (FORTRAN + ALGOL + COBOL)
        - ALGOL 68 (Rigorous successor to ALGOL 60)
        - Pascal (Simple successor to ALGOL 60)
        - Simula (Classes, data abstraction)

Topology of Third-generation Languages

- Applications in these languages exhibit:
    - Support separately compiled module (group subprograms that were most likely to change together)
    - Dismal support for data abstraction and strong typing, hence such errors could be detected only during execution of the program



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

Fourth-generation Languages (1970-1980)

- Major Features
    - **Relational Models for large data handling**
    - **Notions of Programming:**
        - Low-level access for Systems Programming
        - Efficiency
    - **Moving away from "One-language-fits-it-all"**
- Languages
    - C (Efficient, small executable)

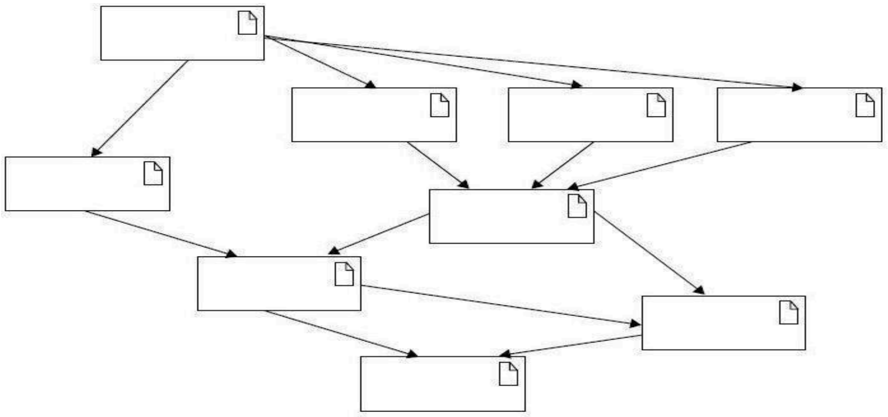
        - FORTRAN 77 (ANSI standardization)
        - SQL (Structured Query Language)

Object-orientation Boom (1980-1990)

- Major Features
    - **Object Models**
    - **Notions of Programming:**
        - Strong Typing
        - Safety
        - Exception
- Languages
    - Smalltalk 80 (Pure object-oriented language)
    - C++ (Derived from C and Simula)
    - Ada83 (Strong typing; heavy Pascal influence)
    - Eiffel (Derived from Ada and Simula)

Topology of Small to Medium Sized Applications Using OB and OOP Languages

- Applications in Object Based (OB) and Object Oriented Programming (OOP) languages exhibit:
    - Fundamental logical building blocks are no longer algorithms, but instead are classes and objects
    - Little or no global data
    - Classes, objects, and modules provide an essential yet insufficient means of abstraction



The Topology of Small to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

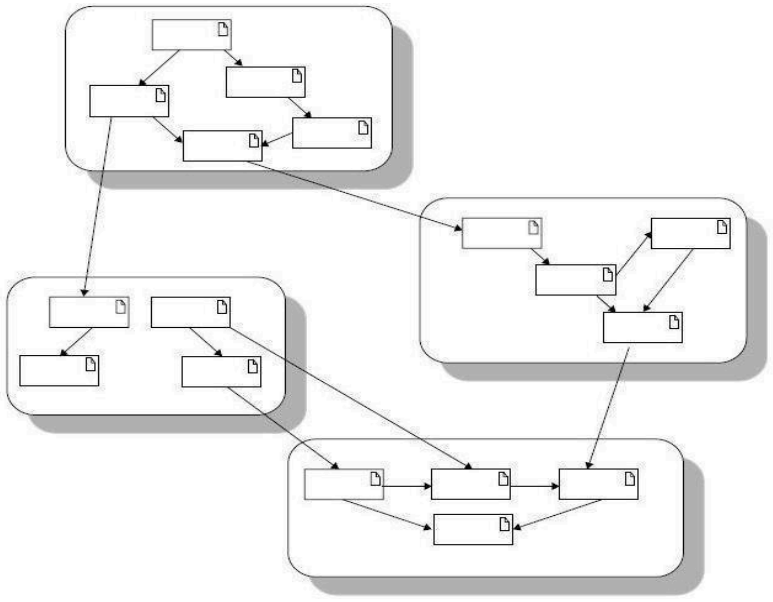
Emergence of Frameworks (1990—today)

- *Major Features*
    - **Programming Frameworks**
    - **Use of OOAD for Large Scale Systems**

- **Notions of Programming:**
    - Dynamic Typing
    - Portability
    - Threads
- **Frameworks:**
    - J2SE, J2EE, J2ME (Java-based frameworks for standard, enterprise & mobile computing)
    - .NET (Microsoft's object-based framework)

- *Languages*
    - Visual Basic (Development GUI for Windows applications)
    - Java (Successor to Oak; designed for portability)
    - Python (00 scripting, portable, dynamically typed)
    - Visual C# (Java competitor for .NET Framework)
    - VB.NET (Visual Basic for NET Framework)

- Applications in these language frameworks exhibit:
    - Programming-in-the-large
    - At any given level of abstraction, meaningful collections of objects achieve some higher-level behavior



The Topology of Small to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

1.8 Foundation of Object Model

Structured design methods evolved to guide developers who use procedural language and algorithms as their fundamental building blocks to build complex system.

Similarly, object-oriented design methods have evolved to help developers who exploit the expressive power of object-based and object-oriented programming languages, using the class and object as basic building blocks.
With Algorithmic Decomposition alone, only limited amount of complexity can be handled — hence we turn to Object-Oriented Decomposition. The following events have contributed to the evolution:

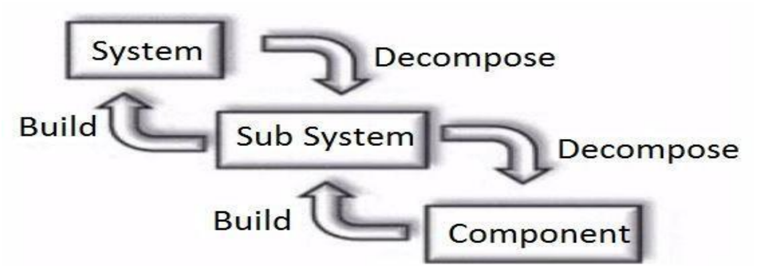
- Advances in computer architecture
- Advances in programming languages
- Advances in programming methodology
- Advances in database models
- Research in artificial intelligence
- Advances in philosophy and cognitive science

Object Oriented Analysis (OOA)
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***
The stages for Object-Oriented Analysis are:

- Identifying Objects
- Identifying Structure
- Identifying Attributes
- Identifying Associations
- Defining Services

Object Oriented Design (OOD)

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 state and dynamic models of the system under design.
There are two important parts to this definition: object-oriented design (1) leads to an object-oriented decomposition and (2) uses different notations to express different models of the logical (class and object structure) and physical (module and process architecture) design of a system, in addition to the static and dynamic aspects of the system.



Object-oriented Decomposition

Object-Oriented Programming

What is object-oriented programming (OOP)? We define it as follows:
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.
There are three important parts to this definition:
(1) Object-oriented programming uses objects, not algorithms, as its fundamental logical building blocks
(2) Each object is an instance of some class; and
(3) Classes may be related to one another via inheritance relationships

1.9 Elements of Object Model

There are four major (mandatory and essential) elements of the object model:

1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy

In addition, there are three minor (useful but not essential) elements of the object model:

1. Typing
2. Concurrency
3. Persistence

Abstraction

Abstraction refers to the process of presenting essential features without including any complex background details. Focus attention on only one aspect of the problem and ignore irrelevant details. It is also called as Model Building. Abstraction focuses on the outside view of an object. Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

Abstraction: Example — Hydroponics Gardening System

- Hydroponics farm:
    - Plants are grown in a nutrient solution, without sand, gravel, or other soils
    - Control diverse factors such as temperature, humidity, light, pH, and nutrient concentrations
    - Design an automated system that constantly monitors and adjusts these elements

    - An automated gardener should efficiently carry out growing plans for the healthy production of multiple crops
- Key Abstractions:
    - Sensors
    - Heaters
    - Growing Plans

Abstraction:	Temperature Sensor
Important Characteristics:	temperature location
Responsibilities:	report current temperature calibrate

Figure: Abstraction of a Temperature Sensor

Encapsulation

Wrapping up of code and data together into a single unit is known as encapsulation. Encapsulation hides the details of the implementation of an object. Encapsulation compartmentalizes the elements of an abstraction that constitute:

- Structure
- Behavior

Encapsulation Example - Hydroponics Gardening System

A heater is at a fairly low level of abstraction, and thus we might decide that there are only three meaningful operations that we can perform on this object: turn it on, turn it off, and find out if it is running. All a client needs to know about the class Heater is its available interface (i.e., the responsibilities that it may execute at the client's request).

Abstraction:	Heater
Important Characteristics:	location status
Responsibilities:	turn on turn off provide status

Related Candidate Abstractions: Heater Controller, Temperature Sensor

Figure: Abstraction of a Heater

Modularity

Modularity refers to partition a program into individual components to manage complexity. The divided modules compiled separately or together, but Modules connected with the other module. Because the solution may not be known when the design stage starts, decomposition into smaller modules may be quite difficult. Modularization should follow the semantic grouping of the common and interrelated functionality of the system.

Objectives of Modularity

- Modular Decomposition
    - o Systematic mechanism to divide problem into individual components
- Modular Composability
    - o Enable reuse of existing components
- Modular Understandability
    - o Understand module as a unit
- Modular Protection
    - o Errors are localized and do not spread to other modules

Modularization: Example Hydroponics Gardening System

Suppose we decide to use a commercially available workstation where the user can control the system's operation. At this workstation, an operator could create new growing plans; modify old ones, a follow the progress of currently active ones. The Key Abstractions are Growing plan, User Interface.

Growing plan

- Create a module whose purpose is to collect all of the classes associated with individual growing plans (for example, FruitGrowingPlan, GrainGrowingPlan)
- The implementations of these GrowingPlan classes would appear in the implementation of this module a

User Interface

- Another module whose purpose is to collect all of the code associated with all user interface functions

Hierarchy

Hierarchy is a ranking or ordering of abstractions. Two most important hierarchies in a complex system are

- Class structure or is-a hierarchy (Abstraction or IS—A)
- Object structure or part-of hierarchy (Decomposition or HAS—A)

The common structure & behavior are migrated to the superclass. Superclass represents a generalized abstraction and subclasses represent specializations. A subclass can add, modify & hide methods from the superclass.

Examples of Hierarchy: Single Inheritance

Inheritance is the most important —is a‖ hierarchy, and it is an essential element of object- oriented systems. Basically, inheritance defines a relationship among classes, wherein one class shares the structure or behavior defined in one or more classes (denoting single inheritance and multiple inheritance, respectively). Inheritance thus represents a hierarchy of abstractions, in which a subclass inherits from one or more super classes. Typically, a subclass augments or redefines the existing structure and behavior of its super classes.

For example, a bear ―is a‖ kind of mammal, a house ―is a‖ kind of tangible asset, and a quick sort
―is a‖ particular kind of sorting algorithm.

Typing
The concept of a type derives primarily from the theories of abstract data types. A type is a precise characterization of structural or behavioral properties which a collection of entities all share. For our purposes, we will use the terms type and class interchangeably. Although the concepts of a type and a class are similar, we include typing as a separate element of the object. A Programming Language may be

- Statically typed
- Dynamically typed
- Strongly typed
- Weakly typed
- Untyped

Polymorphism
Single method exhibiting different behaviours in the same class or different classes is known as polymorphism. Polymorphism is the most powerful feature of object-oriented programming languages next to the support for abstraction.

Concurrency
Concurrency is the property that distinguishes an active object from one that is not active. It allows multiple tasks to execute, interact and collaborate at the same time to achieve the global functionality. Concurrency is critical for Client-Server Model of computation. Concurrency is of two types

- Heavyweight and
- Lightweight Concurrency

Heavyweight Concurrency
A heavyweight process is typically independently managed by the target operating system and so encompasses its own address space. Communication among heavyweight processes is expensive and involves inter-process communication. It is commonly called a Process.

Lightweight Concurrency
A lightweight process lives within a single OS process along with other lightweight processes and shares the same address space. Communication among lightweight processes is less expensive and often involves shared data. It is commonly called a Thread.

1.10 Applying the Object Model
The object model is fundamentally different from the models embraced by the more traditional methods of structured analysis, structured design, and structured programming. 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 hierarchy, relative primitives (i.e., multiple levels of abstraction), separation of concerns, patterns, and stable intermediate forms.

Benefits of the Object Model: Object model introduces several new elements which are advantageous over traditional method of structured programming. The significant benefits are:

- Use of object model helps us to exploit the expressive power of object based and object oriented programming languages. Without the application of elements of object model, more powerful feature of languages such as C++, object pascal, ada are either ignored or greatly misused.
- Use of object model encourages the reuse of software and entire designs, which results in the creation of reusable application framework.
- Use of object model produces systems that are built upon stable intermediate forms, which are more resilient to change.
- Object model appears to the working of human cognition, many people who have no idea how a computer works find the idea of object oriented systems quite natural.