# What Is Analysis and Design?

**Analysis** emphasizes an *investigation* of the problem and requirements, rather than a solution. For example, if a new computerized library information system is desired, how will it be used?

"Analysis" is a broad term, best qualified, as in *requirements analysis* (an investigation of the requirements) or *object analysis* (an investigation of the domain objects).

**Design** emphasizes a *conceptual solution* that fulfills the requirements, rather than its implementation. For example, a description of a database schema and software objects. Ultimately, designs can be implemented.

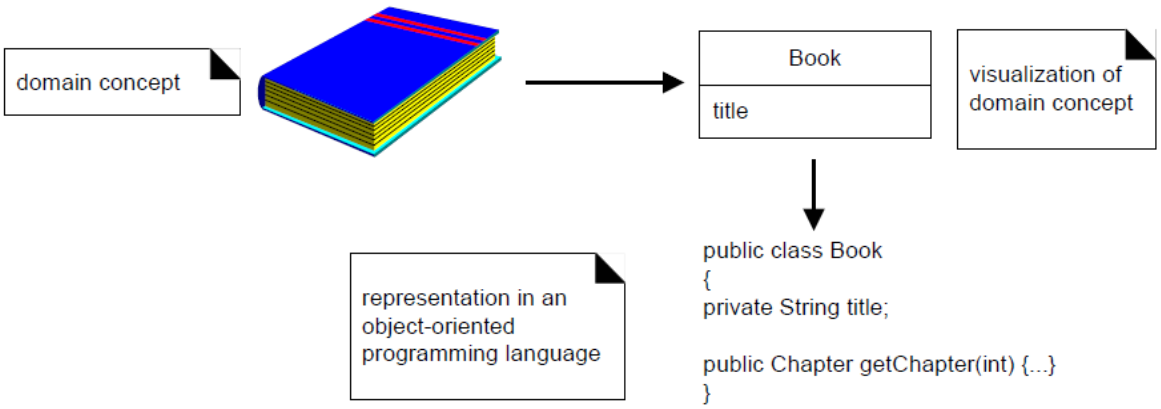
Analysis and design have been summarized in the phase *do the right thing (analysis), and do the thing right (design).*

# What Is Object-Oriented Analysis and Design?

During **object-oriented analysis,** there is an emphasis on finding and describing the objects—or concepts—in the problem domain. For example, in the case of the library information system, some of the concepts include *Book, Library,* and *Patron.*

During **object-oriented design,** there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, in the library system, a *Book* software object may have a *title* attribute and a *getChapter()* method

Finally, during implementation or object-oriented programming, design objects are implemented, such as a *Book* class in Java.



# Object-Oriented Analysis

Object–Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects.

The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions. They are modelled after real-world objects that the system interacts with.

Grady Booch has defined OOA as, *"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 primary tasks in object-oriented analysis (OOA) are −

- Identifying objects
- Organizing the objects by creating object model diagram
- Defining the internals of the objects, or object attributes
- Defining the behavior of the objects, i.e., object actions
- Describing how the objects interact

The common models used in OOA are use cases and object models.

# Object-Oriented Design

Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis. In OOD, concepts in the analysis model, which are technology−independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.

The implementation details generally include −

- Restructuring the class data (if necessary),
- Implementation of methods, i.e., internal data structures and algorithms,
- Implementation of control, and
- Implementation of associations.

Grady Booch has defined object-oriented design as *"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 Programming

Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

The important features of object–oriented programming are −

- Bottom–up approach in program design
- Programs organized around objects, grouped in classes
- Focus on data with methods to operate upon object's data
- Interaction between objects through functions
- Reusability of design through creation of new classes by adding features to existing classes

Some examples of object-oriented programming languages are C++, Java, Smalltalk, Delphi, C#, Perl, Python, Ruby, and PHP.

Grady Booch has defined object–oriented programming as *"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 Model

The object model visualizes the elements in a software application in terms of objects. The basic concepts and terminologies of object–oriented systems are as follows .

## Objects and Classes

The concepts of objects and classes are intrinsically linked with each other and form the foundation of object–oriented paradigm.

### Object

An object is a real-world element in an object–oriented environment that may have a physical or a conceptual existence. Each object has −

- Identity that distinguishes it from other objects in the system.
- State that determines the characteristic properties of an object as well as the values of the properties that the object holds.

- Behavior that represents externally visible activities performed by an object in terms of changes in its state.

Objects can be modelled according to the needs of the application. An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

## Class

A class represents a collection of objects having same characteristics that exhibit common behaviour. It gives the blueprint or description of the objects that can be created from it. Creation of an object as a member of a class is called instantiation. Thus, object is an instance of a class.

The constituents of a class are −

- A set of attributes for the objects that are to be instantiated from the class. Generally, different objects of a class have some difference in the values of the attributes. Attributes are often referred as class data.
- A set of operations that portray the behaviour of the objects of the class. Operations are also referred as functions or methods.

**Example**

Let us consider a simple class, Circle, that represents the geometrical figure circle in a two–dimensional space. The attributes of this class can be identified as follows −

- x–coord, to denote x–coordinate of the center
- y–coord, to denote y–coordinate of the center
- a, to denote the radius of the circle

Some of its operations can be defined as follows −

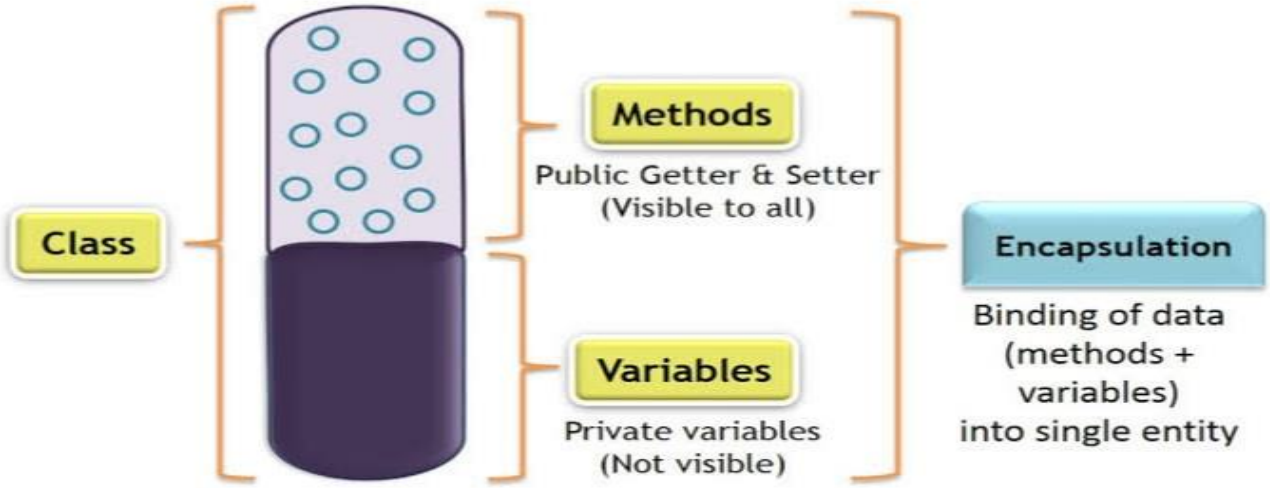
- findArea(), method to calculate area
- findCircumference(), method to calculate circumference
- scale(), method to increase or decrease the radius

During instantiation, values are assigned for at least some of the attributes. If we create an object my_circle, we can assign values like x-coord : 2, y-coord : 3, and a : 4 to depict its state. Now, if the operation scale() is performed on my_circle with a scaling factor of 2, the value of the variable a will become 8. This operation brings a change in the state of my_circle, i.e., the object has exhibited certain behavior.

## Encapsulation

The data of an object is encapsulated within its methods. These data cannot be directly accessed by the outside objects. To access the data internal to an object, other objects have to invoke its methods.

Basically,Encapsulation is the process of binding both attributes and methods together into a single unit within a class. Through encapsulation, the internal details of a class can be hidden from outside.




If required, Encapsulation permits the elements of the class to be accessed from outside only through the interface provided by the class.

Encapsulation offers the following three important advantages:

- **Data Hiding**
- **Protection from Unauthorized access of Object's Data**
- **Weak Coupling**

## Data Hiding

Typically, a class is designed such that its data (attributes) can be accessed only by its class methods and insulated from direct outside access. This process of insulating an object's data is called data hiding or information hiding.

**Example**

In the class Circle, data hiding can be incorporated by making attributes invisible from outside the class and adding two more methods to the class for accessing class data, namely −

- setValues(), method to assign values to x-coord, y-coord, and a

- getValues(), method to retrieve values of x-coord, y-coord, and a

Here the private data of the object my_circle cannot be accessed directly by any method that is not encapsulated within the class Circle. It should instead be accessed through the methods setValues() and getValues().

### Protection from unauthorized data access

The encapsulation feature protects an object's variables from being accidentally corrupted by other objects. This protection includes protection from unauthorized access and also protection from the problems that arise from concurrent access to data such as deadlock and inconsistent values.

### Weak coupling

Since objects do not directly change each others internal data, they are weakly coupled. Weak coupling among objects enhances understandability of the design since each object can be studied and understood in isolation from other objects.

## Message Passing

Any application requires interaction among objects to fulfil its requirement. Objects in a system may communicate with each other using message passing. Suppose a system has two objects: obj1 and obj2. The object obj1 sends a message to object obj2, if obj1 wants obj2 to execute one of its methods.

The features of message passing are −

- Message passing enables all interactions between objects.
- Message passing essentially involves invoking class methods.
- Objects in different processes can be involved in message passing.

## Inheritance

Inheritance is the mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities. The existing classes are called the base classes/parent classes/super-classes, and the new classes are called the derived classes/child classes/subclasses. The subclass can inherit or derive the attributes and methods of the super-class(es) provided that the super-class allows so. Besides, the subclass may add its own attributes and methods and may modify any of the super-class methods. **Inheritance defines an "is – a" relationship.**

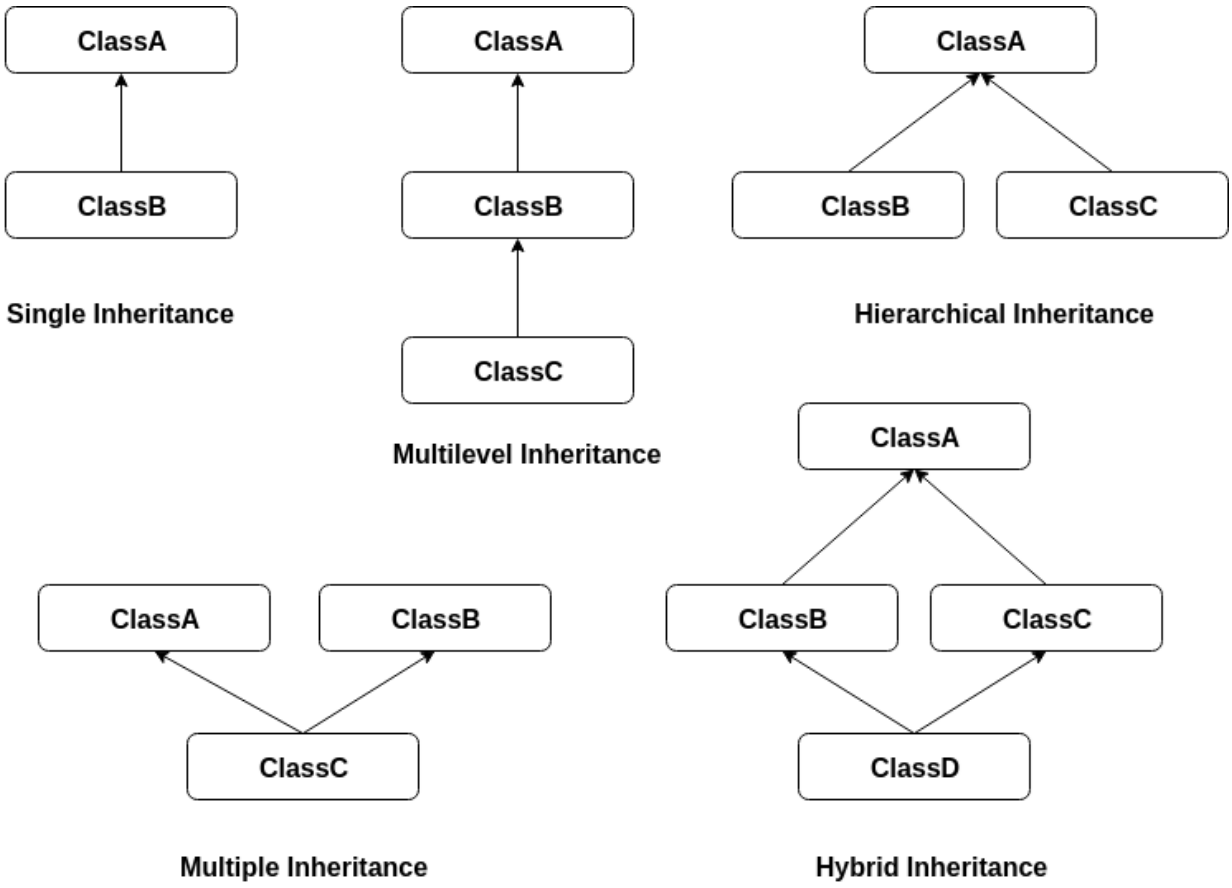
**Example**

From a class Mammal, a number of classes can be derived such as Human, Cat, Dog, Cow, etc. Humans, cats, dogs, and cows all have the distinct characteristics of mammals. In addition, each has its own particular characteristics. It can be said that a cow “is – a” mammal.

## Types of Inheritance

- **Single Inheritance** − A subclass derives from a single super-class.
- **Multiple Inheritance** − A subclass derives from more than one super-classes.
- **Multilevel Inheritance** − A subclass derives from a super-class which in turn is derived from another class and so on.
- **Hierarchical Inheritance** − A class has a number of subclasses each of which may have subsequent subclasses, continuing for a number of levels, so as to form a tree structure.
- **Hybrid Inheritance** − A combination of multiple and multilevel inheritance so as to form a lattice structure.

The following figure depicts the examples of different types of inheritance.



### Advantages of Inheritance

One of the key benefits of inheritance is to minimize the amount of duplicate code in an application by sharing common code amongst several subclasses. Where equivalent code exists in two related classes, the hierarchy can usually be refactored to move the common code up to a mutual superclass. This also tends to result in a better organization of code and smaller, simpler compilation units.

Inheritance can also make application code more flexible to change because classes that inherit from a common superclass can be used interchangeably.

The followings are the common advantages of using Inheritance in OOD.

**Reusability** -- facility to use public methods of base class without rewriting the same

**Extensibility** -- extending the base class logic as per business logic of the derived class

**Data hiding** -- base class can decide to keep some data private so that it cannot be altered by the derived class

**Overriding**-- With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class.

# Polymorphism

Polymorphism is originally a Greek word that means the ability to take multiple forms. In object-oriented paradigm, polymorphism implies using operations in different ways, depending upon the instance they are operating upon. Polymorphism allows objects with different internal structures to have a common external interface. Polymorphism is particularly effective while implementing inheritance.

**Example**

Let us consider two classes, Circle and Square, each with a method findArea(). Though the name and purpose of the methods in the classes are same, the internal implementation, i.e., the procedure of calculating area is different for each class. When an object of class Circle invokes its findArea() method, the operation finds the area of the circle without any conflict with the findArea() method of the Square class.

**There are 2 types of polymorphism which are commonly mentioned**

1. **Static Polymorphism**
2. **Runtime Polymorphism or Dynamic Binding**

## 1. Static Polymorphism

This is also called as **Compile-Time polymorphism, Static binding, Compile-Time binding, Early binding .**

In static binding, the binding of function call to function code is determined at compile time and correspondingly the function or method is called. .

This is actually **method overloading**. Method overloading is having more than one method with the same name but with different arguments (return type may or may not be same). Here when calling the methods compiler choose which method to call depending on the parameters passed when calling. This happens at compile-time.

***Following is a java code fragment to illustrate static or compile time polymorphism.***

```
class Calculator {
    void add(int a, int b) {
         System.out.println(a+b);
    }
    void add(int a, int b, int c) {
         System.out.println(a+b+c);
    }
}


public class Demo {
   public static void main(String args[]) {
       Calculator calculator = new Calculator();

// method with 2 parameters is called

       calculator.add(10, 20);  //output: 30

 // method with 3 parameters is called

       calculator.add(10, 20, 30); //output: 60
   }
}
```

## 2. Dynamic Binding or Runtime Polymorphism

This is also called as **Run-Time polymorphism, Dynamic binding, Run-Time binding, Late binding .**

**Dynamic binding is the process of linking procedure call to a specific sequence of code (method) at run-time. It means that the code to be executed for a specific procedure call is not known until run-time.**

**Dynamic binding is based on two important concepts:**

- **Assignment of an object to a compatible object**
- **Method overriding in a class hierarchy or inheritance**

Lets assume there are methods with same method signature in classes in a class hierarchy (parent-child relationships), these methods are in different forms (this is known as method overriding).

Then when an object is assigned to a class reference and when a method of the object is called, the method in the object's class is executed. Not the method of the reference class (if the reference is a parent class).

What happens here is since the object creation is done at run-time the form of method which should be executed (the method in the object) can be only decided at run-time.

**Runtime Polymorphism in JAVA is called as *Dynamic Method Dispatch* or *Dynamic Dispatch***

## Eg: Java code illustrating Runtime Polymorphism or Dynamic dispatch

```
class Person {
   public void teach(){
      System.out.println("Person can teach");
   }
}
class Teacher extends Person {
   public void teach() {
      System.out.println("Teacher can teach in a school");
   }
}
public class TestTeacher {
   public static void main(String args[]) {
      Person person = new Person();      //Person reference and object

      Person another_person = new Teacher();     //Person reference,
                                                   Teacher object

      Teacher teacher = new Teacher();    //Teacher reference and obj.

      person.teach();          //output: Person can teach

      // Here you can see Teacher object's method is executed even-
      // -though the Person reference was used

       another_person.teach();      //output: Teacher can teach in a
                                              school

        teacher.teach();        //output: Teacher can teach in a school
   }
}
```

## What is Object Orientation ???

The term Object Oriented Means that the software is organized as a collection of discrete objects that incorporates both data structures (data) and behavior (methods)

The object Object Oriented Approach generally includes

- **Identity (i.e. Objects)**
- **Classification (i.e. Classes)**
- **Inheritance**
- **Polymorphism**

**(For Long Question Describe each of these concepts)**

## Benefits of Object Model

Now that we have gone through the core concepts pertaining to object orientation, it would be worthwhile to note the advantages that this model has to offer.

The benefits of using the object model are −

- It helps in faster development of software.
- It is easy to maintain. Suppose a module develops an error, then a programmer can fix that particular module, while the other parts of the software are still up and running.
- It supports relatively hassle-free upgrades.
- It enables reuse of objects, designs, and functions.
- It reduces development risks, particularly in integration of complex systems.

We know that the Object-Oriented Modelling (OOM) technique visualizes things in an application by using models organized around objects. Any software development approach goes through the following stages −

- Analysis,
- Design, and
- Implementation.

In object-oriented software engineering, the software developer identifies and organizes the application in terms of object-oriented concepts, prior to their final representation in any specific programming language or software tools.

# Phases in Object-Oriented Software Development

The major phases of software development using object–oriented methodology are object-oriented analysis, object-oriented design, and object-oriented implementation.

## Object–Oriented Analysis

In this stage, the problem is formulated, user requirements are identified, and then a model is built based upon real–world objects. The analysis produces models on how the desired system should function and how it must be developed. The models do not include any implementation details so that it can be understood and examined by any non–technical application expert.

## Object–Oriented Design

Object-oriented design includes two main stages, namely, system design and object design.

### System Design

In this stage, the complete architecture of the desired system is designed. The system is conceived as a set of interacting subsystems that in turn is composed of a hierarchy of interacting objects, grouped into classes. System design is done according to both the system analysis model and the proposed system architecture. Here, the emphasis is on the objects comprising the system rather than the processes in the system.

### Object Design

In this phase, a design model is developed based on both the models developed in the system analysis phase and the architecture designed in the system design phase. All the classes required are identified. The designer decides whether −

- new classes are to be created from scratch,
- any existing classes can be used in their original form, or
- new classes should be inherited from the existing classes.

The associations between the identified classes are established and the hierarchies of classes are identified. Besides, the developer designs the internal details of the classes and their associations, i.e., the data structure for each attribute and the algorithms for the operations.

## Object–Oriented Implementation and Testing

In this stage, the design model developed in the object design is translated into code in an appropriate programming language or software tool. The databases are created and

the specific hardware requirements are ascertained. Once the code is in shape, it is tested using specialized techniques to identify and remove the errors in the code.

# Principles of Object-Oriented Systems or OO themes

The conceptual framework of object–oriented systems is based upon the object model. This framework is based on the major elements in an object-oriented system −

**Major Elements** − By major, it is meant that if a model does not have any one of these elements, it ceases to be object oriented. The four major elements are −

- **Abstraction**
- **Encapsulation**
- **Modularity**
- **Hierarchy**

# Abstraction

Abstraction means to focus on the essential features of an element or object in OOP, ignoring its extraneous or accidental properties. The essential features are relative to the context in which the object is being used.

Grady Booch has defined abstraction as follows −

“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.”

The main purpose of using the abstraction is to consider only those aspects of a problem that are relevant to a given purpose and to supress or hide all aspects that are not relevant to the problem.

**Abstraction** is supported in two different ways in Object Oriented Design. These are

- **Feature Abstraction**
- **Data Abstraction.**

## Feature abstraction:

A class hierarchy can be viewed as defining several levels (hierarchy) of abstraction, where each class is an abstraction of its subclasses. That is, every class is a simplified (abstract) representation of its derived classes and retains only those features that are

common to all its children classes and ignores the rest of the features. Thus, the inheritance mechanism can be thought of as providing feature abstraction.

### **Data abstraction**:

An object itself can be considered as a data abstraction entity, because it abstracts out the exact way in which it stores its various private data items and it merely provides a set of methods to other objects to access and manipulate these data items. In other words, we can say that data abstraction implies that each object hides (abstracts away) from other objects the exact way in which it stores its internal information.

This helps in developing good quality programs, as it causes objects to have low coupling with each other, since they do not directly access any data belonging to each other.

Each object only provides a set of methods, which other objects can use for accessing and manipulating this private information of the object.

For example, a stack object might store its internal data either in the form of an array of values or in the form of a linked list. Other objects would not know how exactly this object has stored its data (i.e. data is abstracted) and how it internally manipulates its data. What they would know is the set of methods such as push, pop, and top-of-stack that it provides to the other objects for accessing and manipulating the data.

**Note:** An important advantage of the principle of data abstraction is that it reduces coupling among various objects, Therefore, it leads to a reduction of the overall complexity of a design, and helps in easy maintenance and code reuse.

# Encapsulation

***Already discussed. Plaese refer to page 5 and 6***

## Modularity

Modularity is the process of decomposing a problem (program) into a set of modules so as to reduce the overall complexity of the problem. Booch has defined modularity as –

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

Modularity is intrinsically linked with encapsulation. Modularity can be visualized as a way of mapping encapsulated abstractions into real, physical modules having high cohesion within the modules and their inter–module interaction or coupling is low.

## Hierarchy

In Grady Booch's words, "Hierarchy is the ranking or ordering of abstraction". Through hierarchy, a system can be made up of interrelated subsystems, which can have their own subsystems and so on until the smallest level components are reached. It uses the principle of "divide and conquer". Hierarchy allows code reusability.

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.
- **"PART–OF or HAS_A" hierarchy** − It defines the hierarchical relationship in aggregation by which a class may be composed of other classes. For example, Aggregation and Composition.

# Models for Analysis and Design

A ***model*** is an abstraction of something for the purpose of understanding it before building it. Because a model omits nonessential details, it is easier to manipulate than the original entity. Abstraction is a fundamental human capability that permits us to deal with complexity.

To build complex systems,

- the developer must abstract different views of the system,
- build models using precise notations,
- verify that the models satisfy the requirements of the system, and
- gradually add detail to transform the models into an implementation.

The Purpose of Modelling are

- Testing a physical entity before building it
- Communication with customers
- Visualization
- Reduction of complexity

# The Three Models for OOAD

Three kinds of models are used to describe an Object Oriented System from different viewpoints:

- **the Class Model**
- **the State Model**
- **the Inteaction Model**

**The *class model*** represents the static, structural, “data” aspects of a system. **The *state model*** represents the temporal, behavioural, “control” aspects of a system. **The *interaction model*** represents the collaboration of individual objects, the “interaction” aspects of a system.

A typical software procedure incorporates all three aspects: It uses data structures (class model), it sequences operations in time (state model), and it passes data and control among objects (interaction model).

Each model applies during all stages of development and acquires detail as development progresses.

A complete description of a system requires models from all three viewpoints.

## The Class Model

- The ***class model*** describes the structure of objects in a system—their identity, their relationships to other objects, their attributes, and their operations.
- The class model provides context for the state and interaction models.
- Class diagrams express the class model. Generalization lets classes share structure and behaviour, and associations relate the classes.
- Classes define the attributes carried by each object and the operations that each object performs or undergoes.

## The State Model

- The ***state model*** describes the aspects of an object that change over time.
- The ***state model*** describes those aspects of objects concerned with time and the sequencing of operations, events that mark changes, states that define the context for events, and the organization of events and states.
- The state model captures ***control,*** the aspect of a system that describes the sequences of operations that occur, without regard for what the operations do, what they operate on, or how they are implemented.
- **State diagrams express the state model**. Each state diagram shows the state and event sequences. permitted in a system for one class of objects.
- A ***state diagram*** is a graph whose nodes are states and whose arcs are transitions between states caused by events.
- State diagrams also refer to the other models. Actions and events in a state diagram become operations on objects in the class model. References between state diagrams become interactions in the interaction model.

## The **interaction model**

- ***The Interaction Model*** describes how the objects in a system cooperate to achieve broader results.
- The ***interaction model*** describes interactions between objects—how individual objects collaborate to achieve the behaviour of the system as a whole.
- **Use cases, sequence diagrams, and activity diagrams document the interaction model.**
- The interaction model starts with use cases that are then elaborated with sequence
- and activity diagrams
- A ***use case*** focuses on the functionality of a system—that is, what a system does for users. Use cases document major themes for interaction between the system and outside actors.

- **Sequence diagrams** show the objects that interact and the time sequence of their interactions.

- An ***activity diagram*** elaborates important processing steps and the flow of control among the processing steps of a computation.

The three models are separate parts of the description of a complete system but are cross-linked. Each model describes one aspect of the system but contains references to the other models. The class model describes data structure on which the state and interaction models operate. The operations in the class model correspond to events and actions. The state model describes the control structure of objects. It shows decisions that depend on object values and causes actions that change object values and state. The interaction model focuses on the exchanges between objects and provides a complete overview of the operation of a system.