OBJECT ORIENTED SOFTWARE ENGINEERING

UNIT-1

Introduction

Overview of Object-Orientation

A few concepts that must be understood before working with object-oriented programming and ABL are classes, types, data members, methods and objects. All these concepts are closely related. It is important to understand the difference between these terms. The following list explains all the above important concepts.

- **Type:** A type is a name that identifies specific members of a class, which can include methods, properties, data members, and events.
- **Class:** A class defines the implementation of a type and its class members. An abstract class is essentially a type with an incomplete implementation. Classes in ABL support a basic set of object-oriented concepts.
- **Interface:** An interface also defines a type that identifies certain class members (properties, methods, or events) that a class must implement.
- **Data Members:** Data of a class is stored in data members. Each data member is defined by a name and type.
- **Object:** An object is an instance of a class whose type can be represented as any class or interface that contributes members defined in the object's class hierarchy. Objects have a life cycle in which they can be repeatedly created, used, and destroyed during an ABL session.

Basic Concepts of Object-Orientation

Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Class

The building block of any programming languages that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user-defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions define the properties and behaviour of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage etc and member functions can apply brakes, increase speed etc.

We can say that a **Class** is a blue-print representing a group of objects which shares some common properties and behaviours.

Object

An Object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Object take up space in memory and have an associated address like a record in pascal or structure or union in C.

When a program is executed the objects interact by sending messages to one another.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

Encapsulation

In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section".

Encapsulation also leads to *data abstraction or hiding*. As using encapsulation also hides the data. In the above example, the data of any of the section like sales, finance or accounts are hidden from any other section.

Abstraction

Data abstraction is one of the most essential and important features of object-oriented programming in an Object-oriented programming language. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

- *Abstraction using Classes*: We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
- *Abstraction in Header files*: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Polymorphism

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

C++ supports operator overloading and function overloading.

- *Operator Overloading*: The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.
- *Function Overloading*: Function overloading is using a single function name to perform different types of tasks.
  Polymorphism is extensively used in implementing inheritance.

Example: Suppose we have to write a function to add some integers, some times there are 2 integers, some times there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.

Inheritance

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class**: The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class**: The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability**: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.

Association

Association is a relationship between two objects. In other words, association defines the multiplicity between objects. You may be aware of one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects. Aggregation is a special form of association. Composition is a special form of aggregation.

Aggregation

Aggregation is a special case of association. A directional association between objects. When an object ‘has-a’ another object, then you have got an aggregation between them. Direction between them specified which object contains the other object. Aggregation is also called a “Has-a” relationship.

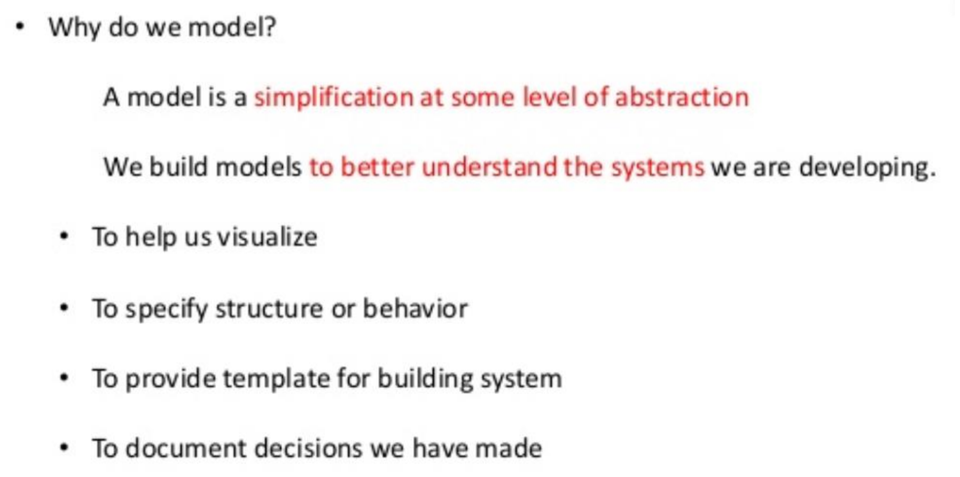
Composition

Composition is a special case of aggregation. In a more specific manner, a restricted aggregation is called composition. When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition.

Difference between aggregation and composition

Composition is more restrictive. When there is a composition between two objects, the composed object cannot exist without the other object. This restriction is not there in aggregation. Though one object can contain the other object, there is no condition that the composed object must exist. The existence of the composed object is entirely optional. In both aggregation and composition, direction is must. The direction specifies, which object contains the other object.

Importance of modeling



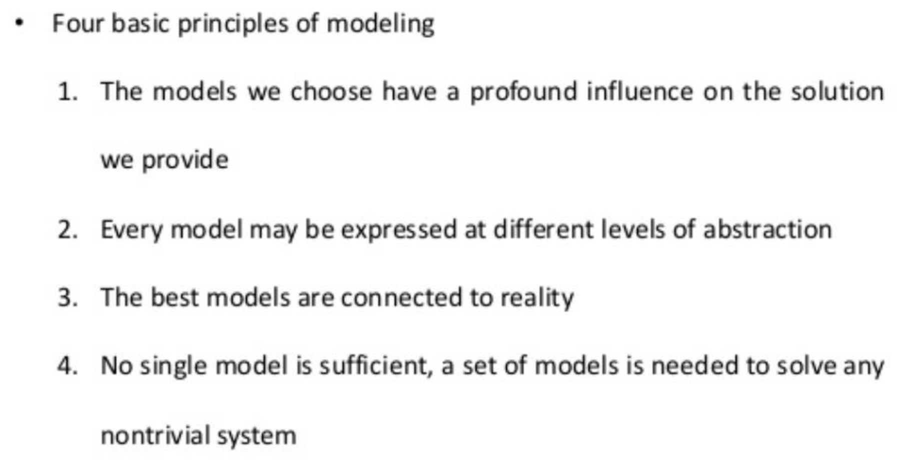
- Why do we model?

  A model is a simplification at some level of abstraction

  We build models to better understand the systems we are developing.

  - To help us visualize
  - To specify structure or behavior
  - To provide template for building system
  - To document decisions we have made

Principles of modeling



- Four basic principles of modeling

  1. The models we choose have a profound influence on the solution we provide
  2. Every model may be expressed at different levels of abstraction
  3. The best models are connected to reality
  4. No single model is sufficient, a set of models is needed to solve any nontrivial system

Object Oriented Modeling

Object-oriented modeling (**OOM**) is an approach to modeling an application that is used at the beginning of the software life cycle when using an object-oriented approach to software development.

The software life cycle is typically divided up into stages going from abstract descriptions of the problem to designs then to code and testing and finally to deployment. Modeling is done at the beginning of the process. The reasons to model a system before writing the code are:

- Communication. Users typically cannot understand programming language or code. Model diagrams can be more understandable and can allow users to give developers feedback on the appropriate structure of the system. A key goal of the Object-Oriented approach is to decrease the "semantic gap" between the system and the real world by using terminology that is the same as the functions that users perform. Modeling is an essential tool to facilitate achieving this goal .
- Abstraction. A goal of most software methodologies is to first address "what" questions and then address "how" questions. I.e., first determine the functionality the system is to provide without consideration of implementation constraints and then consider how to take this abstract description and refine it into an implementable design and code given constraints such as technology and budget. Modeling enables this by allowing abstract descriptions of processes and objects that define their essential structure and behavior.

Object-oriented modeling is typically done via use cases and abstract definitions of the most important objects. The most common language used to do object-oriented modeling is the Object Management Group's Unified Modeling Language (UML).

OO Life cycle

Object Oriented analysis

In the system analysis or object-oriented analysis phase of software development, the system requirements are determined, the classes are identified and the relationships among classes are identified.

The three analysis techniques that are used in conjunction with each other for object-oriented analysis are object modelling, dynamic modelling, and functional modelling.

Object Modelling

Object modelling develops the static structure of the software system in terms of objects. It identifies the objects, the classes into which the objects can be grouped into and the relationships between the objects. It also identifies the main attributes and operations that characterize each class.

The process of object modelling can be visualized in the following steps −

- Identify objects and group into classes
- Identify the relationships among classes
- Create user object model diagram
- Define user object attributes
- Define the operations that should be performed on the classes
- Review glossary

Dynamic Modelling

After the static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined. This is the purpose of dynamic modelling.

Dynamic Modelling can be defined as "a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world".

The process of dynamic modelling can be visualized in the following steps −

- Identify states of each object
- Identify events and analyze the applicability of actions
- Construct dynamic model diagram, comprising of state transition diagrams
- Express each state in terms of object attributes
- Validate the state–transition diagrams drawn

Functional Modelling

Functional Modelling is the final component of object-oriented analysis. The functional model shows the processes that are performed within an object and how the data changes as it moves between methods. It specifies the meaning of the operations of object modelling and the actions of dynamic modelling. The functional model corresponds to the data flow diagram of traditional structured analysis.

The process of functional modelling can be visualized in the following steps −

- Identify all the inputs and outputs
- Construct data flow diagrams showing functional dependencies
- State the purpose of each function
- Identify constraints
- Specify optimization criteria

Structured Analysis vs. Object Oriented Analysis

The Structured Analysis/Structured Design (SASD) approach is the traditional approach of software development based upon the waterfall model. The phases of development of a system using SASD are −

- Feasibility Study
- Requirement Analysis and Specification
- System Design
- Implementation
- Post-implementation Review

Now, we will look at the relative advantages and disadvantages of structured analysis approach and object-oriented analysis approach.

Advantages/Disadvantages of Object Oriented Analysis

Advantages	Disadvantages
Focuses on data rather than the procedures as in Structured Analysis.	Functionality is restricted within objects. This may pose a problem for systems which are intrinsically procedural or computational in nature.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	It cannot identify which objects would generate an optimal system design.

The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	The object-oriented models do not easily show the communications between the objects in the system.
It allows effective management of software complexity by the virtue of modularity.	All the interfaces between the objects cannot be represented in a single diagram.
It can be upgraded from small to large systems at a greater ease than in systems following structured analysis.	

Advantages/Disadvantages of Structured Analysis

Advantages	Disadvantages
As it follows a top-down approach in contrast to bottom-up approach of object-oriented analysis, it can be more easily comprehended than OOA.	In traditional structured analysis models, one phase should be completed before the next phase. This poses a problem in design, particularly if errors crop up or requirements change.
It is based upon functionality. The overall purpose is identified and then functional decomposition is done for developing the software. The emphasis not only gives a better understanding of the system but also generates more complete systems.	The initial cost of constructing the system is high, since the whole system needs to be designed at once leaving very little option to add functionality later.
The specifications in it are written in simple English language, and hence can be more easily analyzed by non-technical personnel.	It does not support reusability of code. So, the time and cost of development is inherently high.

Modeling and design

The Object-Oriented approach of Building Systems takes the objects as the basis. For this, first the system to be developed is observed and analyzed and the requirements are defined as in any other method of system development. Once this is often done, the objects in the required system are identified. For example, in the case of a Banking

System, a customer is an object, a chequebook is an object, and even an account is an object.

Object-oriented model employs an object-oriented strategy. The primary objectives are:

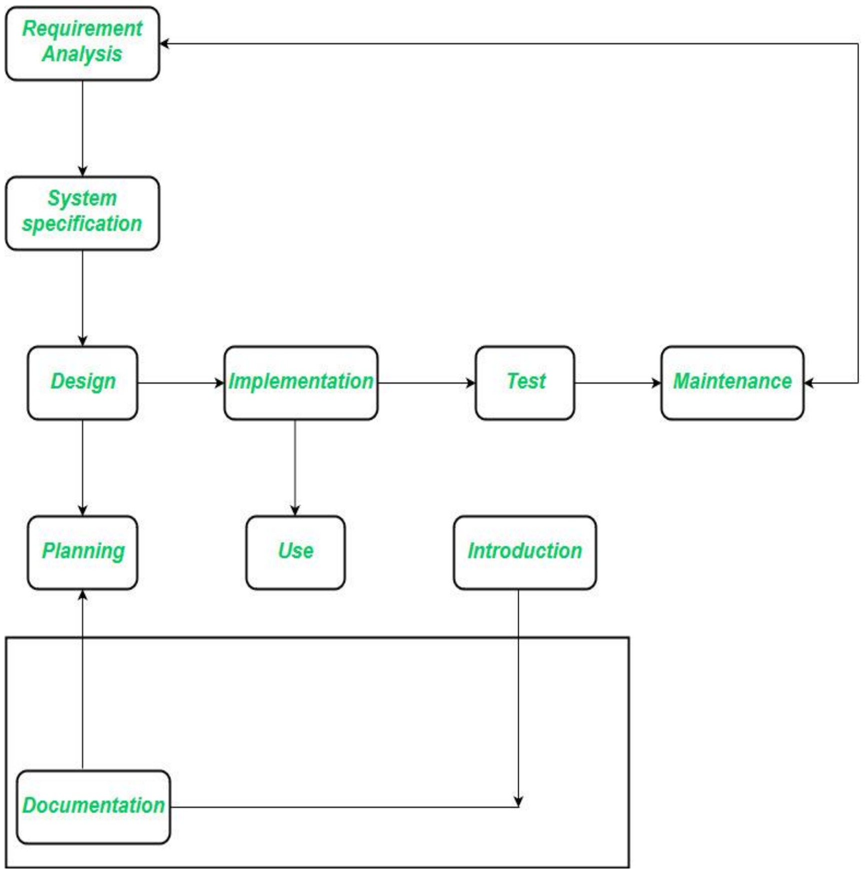
```
1. Object-oriented analysis,
2. Object-oriented design,
3. Object-oriented programming
```

Object-oriented analysis develops an object-oriented model of the application domain.
Object-oriented design develops an object-oriented model of the software system.
Object oriented programming realizes the software design with an object-oriented programming language that supports direct implementation of objects, classes, and inheritance.

There are a variety of object-oriented methodologies such as:

- **Object Identification:**
  System objects and their characteristics and events.
- **Object Organization:**
  Shows how objects are related via "part-of" relationships.
- **Object Interfaces:**
  Shows how objects interact with other objects.

These activities tend to be overlapping and in general and parallel.



The requirements analysis stage strives to achieve an understanding of the client's application domain. The task that a software solution must address emerge in the course of requirement analysis. The requirement analysis phase remains completely independent of an implementation technique that might be applied later.

In the system specification section, the wants definition describes what the software product must do, but not how this goal is to be achieved.

One point of divergence from conventional phase model arises because implementation with object-oriented programming is marked by the assembly of already existing components. The class library serves as a tool that extends beyond the scope of an individual project because class provided by one project can increase productivity in subsequent projects.

Advantages of Object-Oriented Life Cycle Model:

- Design is no longer carried out independently of the later implementation because during the design phase we must consider which components are available for the solution of the problem.
- Design and implementation become more closely associated.
- Duration of the implementation phase is reduced.
- A new job title emerges, the class librarian, who is responsible for ensuring the efficient usability of the class library.

Requirement Elicitation

In requirements engineering, **requirements elicitation** is the practice of researching and discovering the requirements of a system from users, customers, and other stakeholders.[1] The practice is also sometimes referred to as "**requirement gathering**".

The term elicitation is used in books and research to raise the fact that good requirements cannot just be collected from the customer, as would be indicated by the name requirements gathering. Requirements elicitation is non-trivial because you can never be sure you get all requirements from the user and customer by just asking them what the system should do or not do (for Safety and Reliability). Requirements elicitation practices include interviews, questionnaires, user observation, workshops, brainstorming, use cases, role playing and prototyping.

Before requirements can be analyzed, modeled, or specified they must be gathered through an elicitation process. Requirements elicitation is a part of the requirements engineering process, usually followed by analysis and specification of the requirements.

Commonly used elicitation processes are the stakeholder meetings or interviews.[2] For example, an important first meeting could be between software engineers and customers where they discuss their perspective of the requirements.

Introduction to Object Oriented Methodologies

What is Object Oriented Methodology?

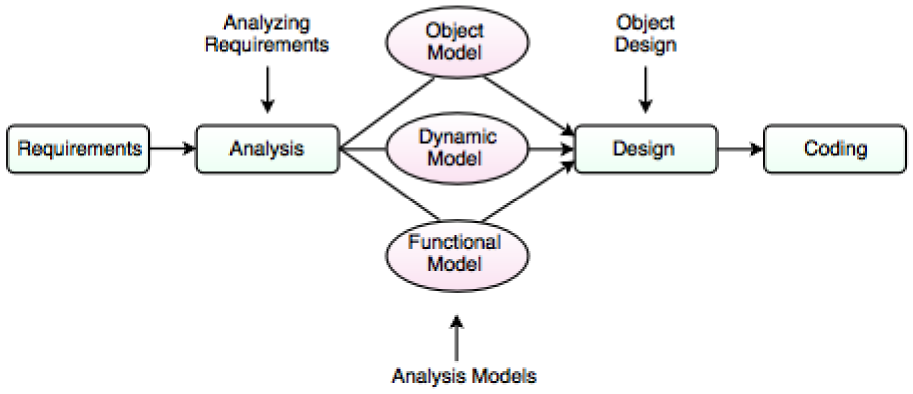
- It is a new system development approach, encouraging and facilitating re-use of software components.
- It employs international standard Unified Modeling Language (UML) from the Object Management Group (OMG).
- Using this methodology, a system can be developed on a component basis, which enables the effective re-use of existing components, it facilitates the sharing of its other system components.
- Object Oriented Methodology asks the analyst to determine what the objects of the system are?, What responsibilities and relationships an object has to do with the other objects? and How they behave over time?

There are three types of Object Oriented Methodologies

1. Object Modeling Techniques (OMT)
2. Object Process Methodology (OPM)
3. Rational Unified Process (RUP)

1. Object Modeling Techniques (OMT)

- It was one of the first object oriented methodologies and was introduced by Rumbaugh in 1991.
- OMT uses three different models that are combined in a way that is analogous to the older structured methodologies.



a. Analysis

- The main goal of the analysis is to build models of the world.

- The requirements of the users, developers and managers provide the information needed to develop the initial problem statement.

b. OMT Models

I. Object Model

- It depicts the object classes and their relationships as a class diagram, which represents the static structure of the system.
- It observes all the objects as static and does not pay any attention to their dynamic nature.

II. Dynamic Model

- It captures the behavior of the system over time and the flow control and events in the Event-Trace Diagrams and State Transition Diagrams.
- It portrays the changes occurring in the states of various objects with the events that might occur in the system.

III. Functional Model

- It describes the data transformations of the system.
- It describes the flow of data and the changes that occur to the data throughout the system.

c. Design

- It specifies all of the details needed to describe how the system will be implemented.
- In this phase, the details of the system analysis and system design are implemented.
- The objects identified in the system design phase are designed.

2. Object Process Methodology (OPM)

- It is also called as second generation methodology.
- It was first introduced in 1995.
- It has only one diagram that is the Object Process Diagram (OPD) which is used for modeling the structure, function and behavior of the system.
- It has a strong emphasis on modeling but has a weaker emphasis on process.
- It consists of three main processes:

I. Initiating: It determines high level requirements, the scope of the system and the resources that will be required.

II. Developing: It involves the detailed analysis, design and implementation of the system.

III. Deploying: It introduces the system to the user and subsequent maintenance of the system.

3. Rational Unified Process (RUP)

- It was developed in Rational Corporation in 1998.
- It consists of four phases which can be broken down into iterations.

  I. Inception

  II. Elaboration

  III. Construction

  IV. Transition
- Each iteration consists of nine work areas called disciplines.
- A discipline depends on the phase in which the iteration is taking place.
- For each discipline, RUP defines a set of artefacts (work products), activities (work undertaken on the artefacts) and roles (the responsibilities of the members of the development team).

Objectives of Object Oriented Methodologies

- To encourage greater re-use.
- To produce a more detailed specification of system constraints.
- To have fewer problems with validation (Are we building the right product?).

Benefits of Object Oriented Methodologies

1. It represents the problem domain, because it is easier to produce and understand designs.

2. It allows changes more easily.

3. It provides nice structures for thinking, abstracting and leads to modular design.

4. Simplicity:

- The software object's model complexity is reduced and the program structure is very clear.

5. Reusability:

- It is a desired goal of all development process.
- It contains both data and functions which act on data.
- It makes easy to reuse the code in a new system.
- Messages provide a predefined interface to an object's data and functionality.

6. Increased Quality:

- This feature increases in quality is largely a by-product of this program reuse.

7. Maintainable:

- The OOP method makes code more maintainable.
- The objects can be maintained separately, making locating and fixing problems easier.

8. Scalable:

- The object oriented applications are more scalable than structured approach.
- It makes easy to replace the old and aging code with faster algorithms and newer technology.

9. Modularity:

- The OOD systems are easier to modify.
- It can be altered in fundamental ways without ever breaking up since changes are neatly encapsulated.

10. Modifiability:

- It is easy to make minor changes in the data representation or the procedures in an object oriented program.

11. Client/Server Architecture:

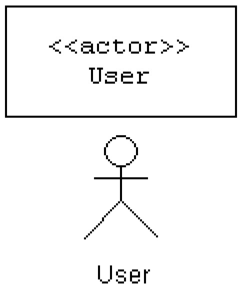
- It involves the transmission of messages back and forth over a network.

Requirements Model-Action & Use cases

Requirements Modeling

Actors

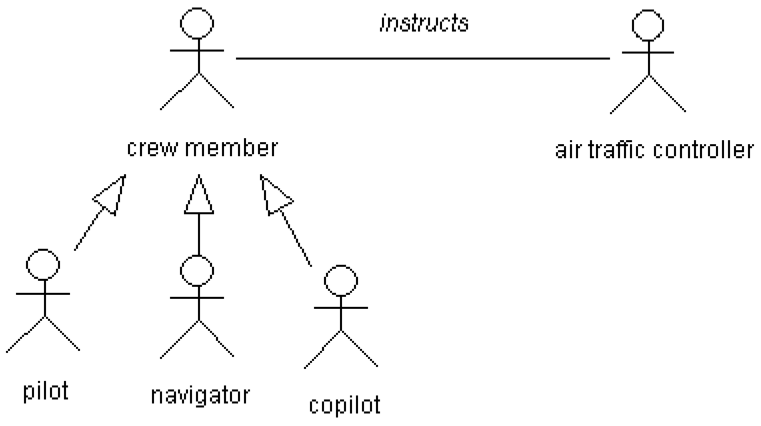
The immediate environment of a software system consists of the users, devices, and programs that the system interacts with. These are called actors.



Types of actors include:

```
users
database systems
clients and servers
platforms
devices
```

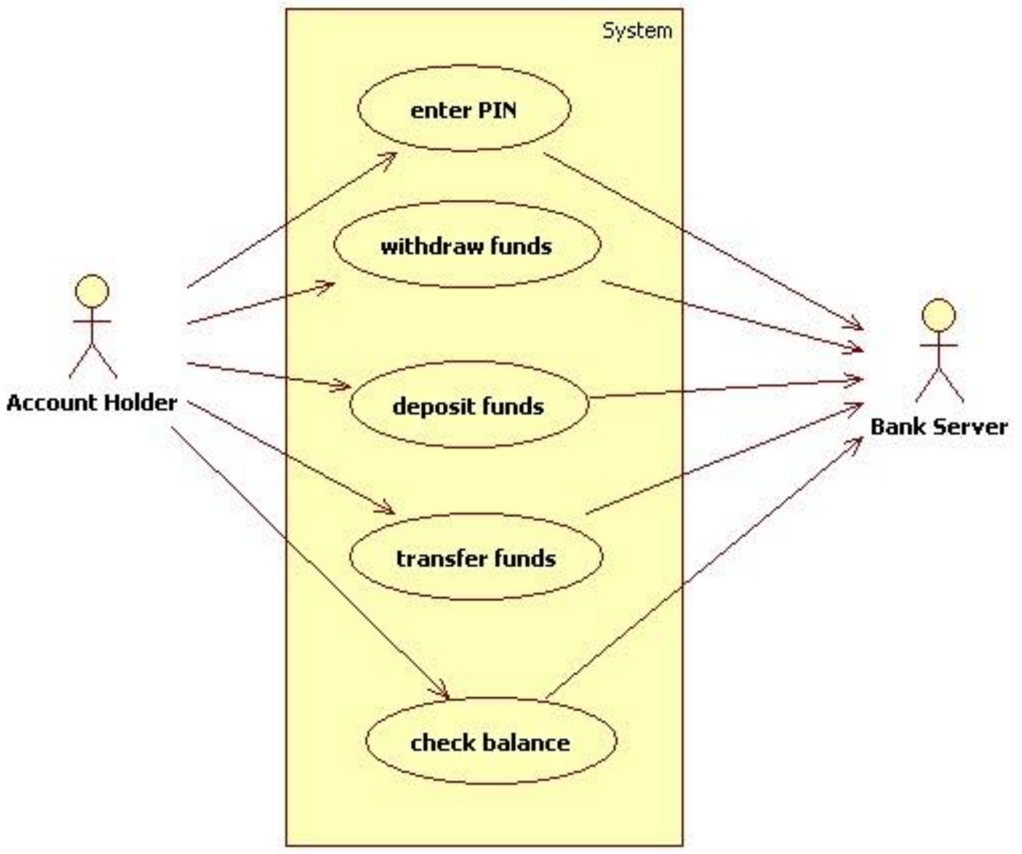
Since actors are classes, we can model relationships between them using class diagram notation:



<u>Use Cases</u>

An actor-system interaction has a goal. These goals are called use cases. Use cases are ellipses. An arrow between a use case and an actor is called an actor association. An actor association represents a conversation between an actor and the use case.

<u>Example: ATM use cases:</u>



Primary actors usually initiate a use case. This is shown by an arrow running from the actor to the use case. Secondary actors respond to use cases.

Use Case Elaborations

A use case diagram by itself is useless. Each use case should be associated with a use case elaboration. A use case elaboration consists of the following information:

```
Name:
Actors:
Description:
Priority: (high, medium, low)
Risk: (high, medium, low)
Scenarios:
�� Scenario 1: this is usually the main scenario
�� Scenario 2: an alternate scenario
�� Scenario 3: another alternate scenario
�� ...
```

For example:

```
Name: Transfer Funds
Actors: Account Holder, Bank Server
Description: The account holder transfers a specified amount of money from
```

```
a specified source account to a specified destination account.
Priority: high
Risk: medium
Scenarios:
�� Scenario 1: Funds are transferred successfully
����� ...
�� Scenario 2: Insufficient funds in source account
����� ...
�� Scenario 3: Source or destination accounts are in use
����� ...
�� Scenario 4: Invalid PIN
����� ...
```

Use Case Scenarios

A use case scenario is a script that describes a typical conversation between a use case and its associated actors.

The main scenario is the most typical conversation in which the goal is achieved. Secondary scenarios describe conversations in which errors or other unusual conditions arise.

Modeling scenarios using scripts

For example, the main scenario for "Transfer Funds" begins by validating the account holder:

```
holder:� Inserts card
ATM: ��� Reads card
ATM: ��� Requests holder info
bank:��� Returns holder info
ATM:���� Requests pin
holder:� Enters PIN
ATM: ��� Validates PIN
ATM: ��� Displays menu
```

Since the main scenario is usually successful, we may assume the PIN was valid. Next the ATM gathers information from the holder about the transaction:

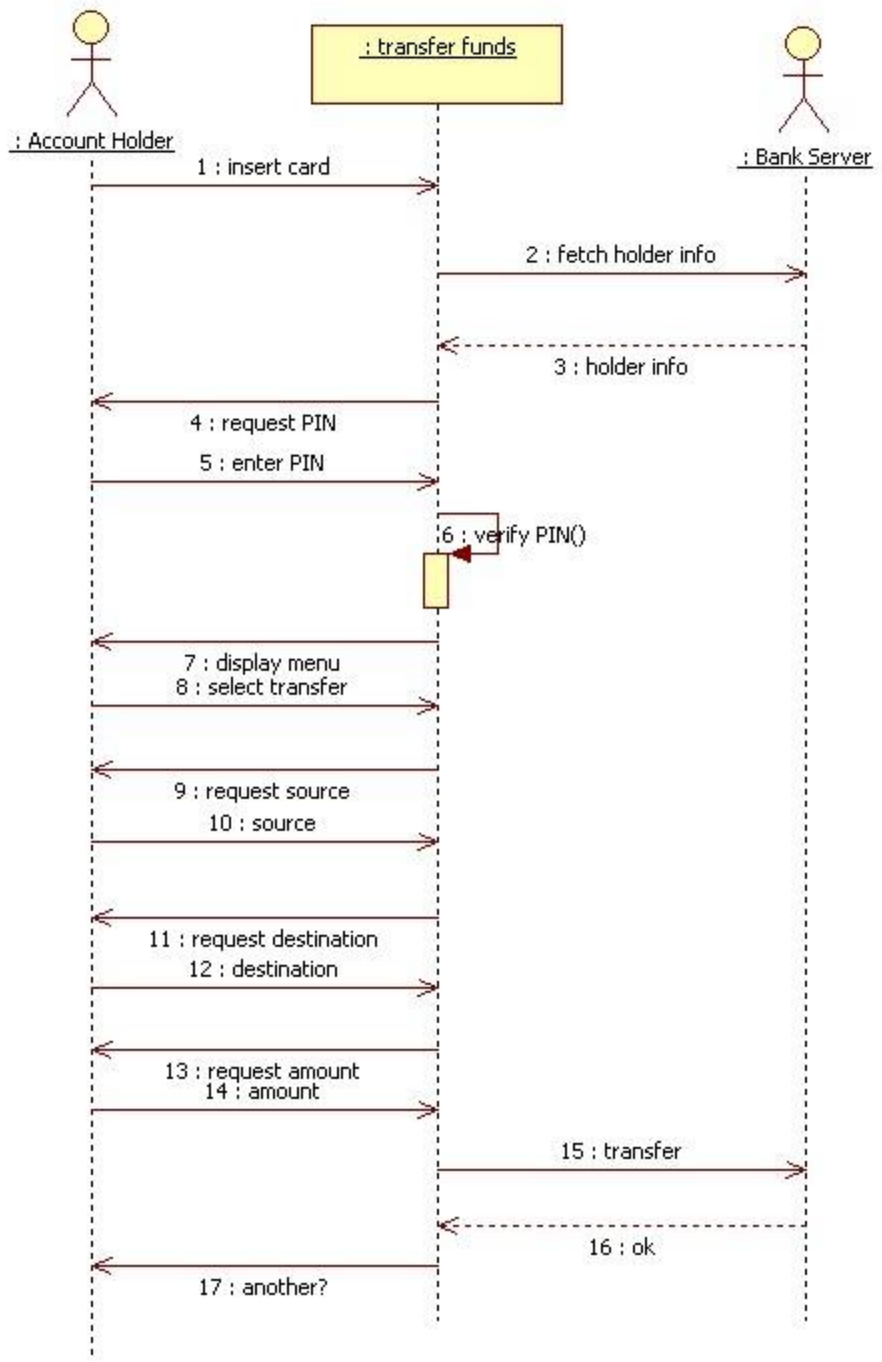
```
holder:� Selects "Transfer"
ATM: ��� Requests source account
holder: Selects source account
ATM:���� Requests destination account
holder:� Selects destination account
ATM:���� Requests amount
holder:� Specifies amount
ATM:���� Requests transfer
bank:��� Transfers funds
bank:��� Acknowledges transfer
```

Finally, the ATM acknowledges the transaction, prints a receipt, and displays the options menu:

```
ATM:���� Acknowledges transaction
ATM:���� Need receipt?
holder:� Selects "yes"
ATM:���� Prints receipt
ATM:���� Displays menu
```

Modeling scenarios using sequence diagrams

Sequence diagrams can also be used to model scenarios:



Use Case Relationships

A use case scenario can be seen as a sequence of actions. An action usually involves an actor or a use case receiving an input from a previous action, doing some processing, and generating some output to a following action.

Assume a main scenario is the following sequence of actions:

```
A1, A2, A3, A4, A5, A6, A7, A8, A9
```

For example:

```
A1 = holder:�� Inserts card
A2 = ATM: ���� Reads card
A3 = ATM: ���� Requests holder info
A4 = bank:���� Returns holder info
A5 = ATM:����� Requests pin
A6 = holder:�� Enters PIN
A7 = ATM: ���� Validates PIN
A8 = ATM: ���� Displays menu
```

An alternate scenario usually shares the same prefix as the main scenario, but has a different suffix:

```
A1, A2, A3, A4, A5, A6, B1, B2
```

For example, if there are insufficient funds, the suffix of Transfer Funds might look like this:

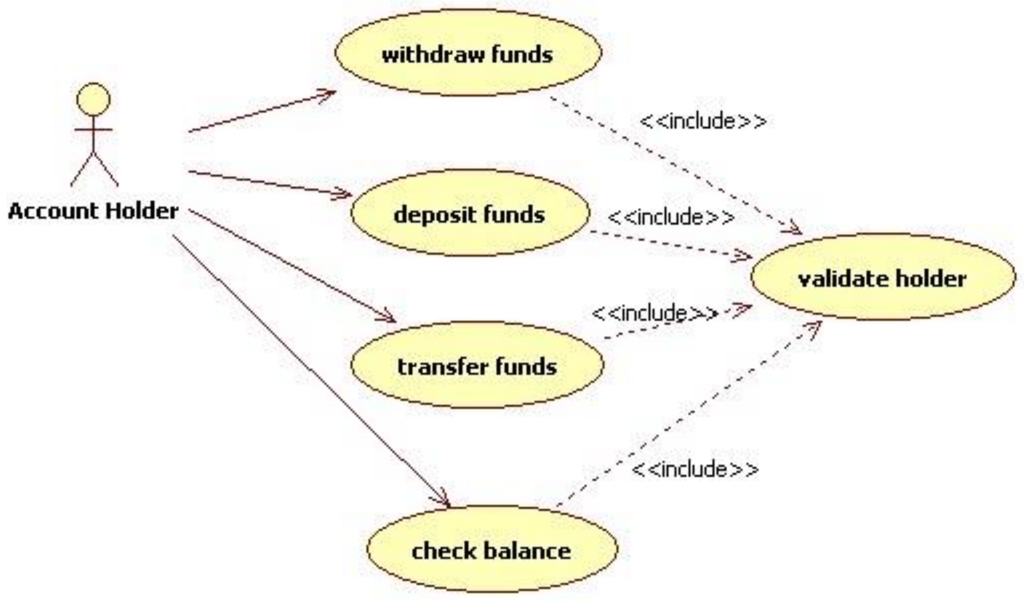
```
holder:� Selects "Transfer"
ATM: ��� Requests source account
holder: Selects source account
ATM:���� Requests destination account
holder:� Selects destination account
ATM:���� Requests amount
holder:� Specifies amount
ATM:���� Requests transfer
bank:��� Detects insufficient funds
bank:��� Transfer Fails
ATM:���� Displays "Insufficient Funds" error
ATM:���� Displays menu
```

Included Use Cases

Sometimes an action sequence appears as a subsequence in many scenarios. In this case we can treat the subsequence as a separate use case and include it in other use cases. For example, most of the ATM use cases begin by validating the holder:

```
holder:� Inserts card
ATM: ��� Reads card
ATM: ��� Requests holder info
bank:��� Returns holder info
ATM:���� Requests pin
holder:� Enters PIN
ATM: ��� Validates PIN
ATM: ��� Displays menu
```

We can create a "Validate User" use case and allow other use cases to explicit invoke it. This is modeled by the "include" arrow:



Use Case Controllers

An included use case is often implemented as an explicit call to another use case. For example, assume all use cases are implemented as instances of classes implementing a use case controller interface:

```
interface UseCase {
�� void execute(Model context) throws Exception;
}
```

Model is any facade that provides access to the application logic and data:

```
interface Model extends Serializable { }
```

For example:

```
class Bank implements Model { ... }
```

We implement the "Validate Holder" use case as follows:

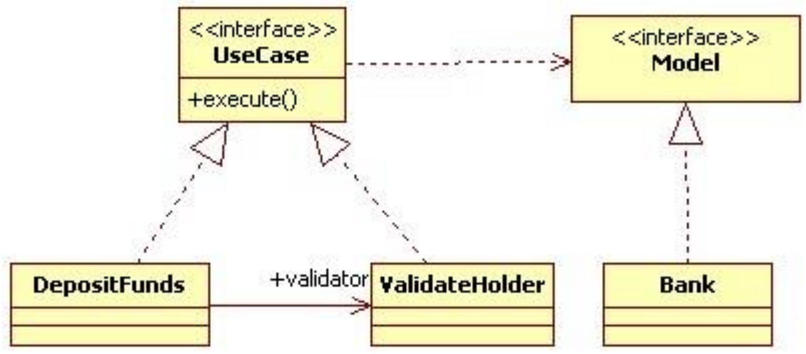
```
class ValidateHolder implements UseCase {
�� public void execute(Model context) throws Exception {
����� // read card & PIN
����� // throw exception if they don't match
�� }
}
```

The "Deposit Funds" use case contains a reference to a "validate Holder" use case, which it explicitly invokes:

```
class DepositFunds implements UseCase {
�� private ValidateHolder validater = new ValidateHolder();
�� public void execute(Model context) throws Exception {
```

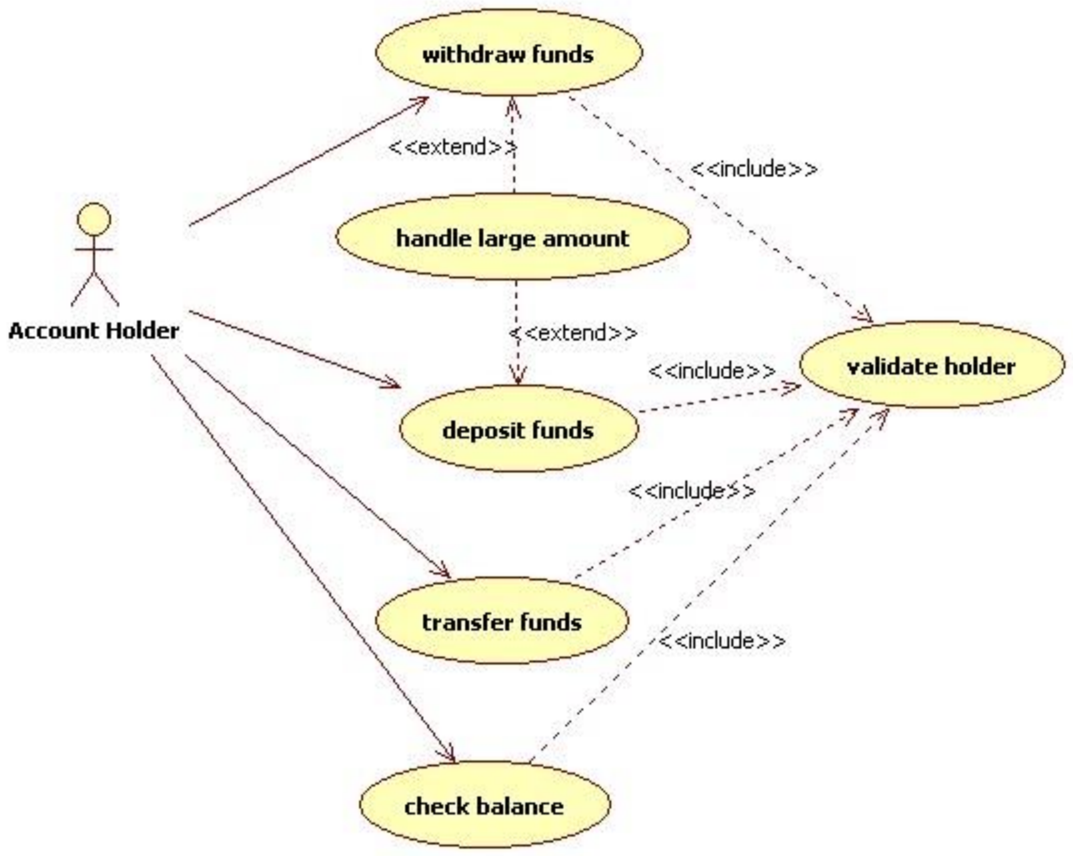
```
����� validater.execute(context);
����� Account destination = getAccount();
����� Money amount = getAmount();
����� destination.deposit(amount);
�� }
}
```

We can model this design with a class diagram:



Extended Use Cases

UML also allows the possibility of an implicit invocation of another use case. For example, assume the bank may want to treat large deposits and withdrawals specially using a pluggable policy. In other words, the bank wants to be able to change the policy without modifying the original deposit and withdraw use cases. This can be done by introducing the policy as a separate use case that connects to the Deposit and Withdraw use cases using an "extend" arrow:



Extended use cases can be implemented by polymorphism. For example, DepositFunds only knows the large amount handler is something the implements the UseCase interface. The actual object can be set and changed dynamically without altering the DepositFunds use case code:

```
class DepositFunds implements UseCase {
�� private ValidateHolder validater = new ValidateHolder();
�� private UseCase amtHandler;
�� private static final Money CRITICAL = new Money(10000);
�� public setLargeAmountHandler(UseCase handler) {
����� amtHandler = handler;
�� }
�� public void execute(Model context) throws Exception {
����� validater.execute(context);
����� Account destination = getAccount();
����� Money amount = getAmount();
����� if (largeAmountHandler != null && amount <= CRITICAL) {
�������� amtHandler.execute(context);
�������� return;
����� }
�� �� destination.deposit(amount);
�� }
}
```

Assume we have a use case controller for handling large amounts:

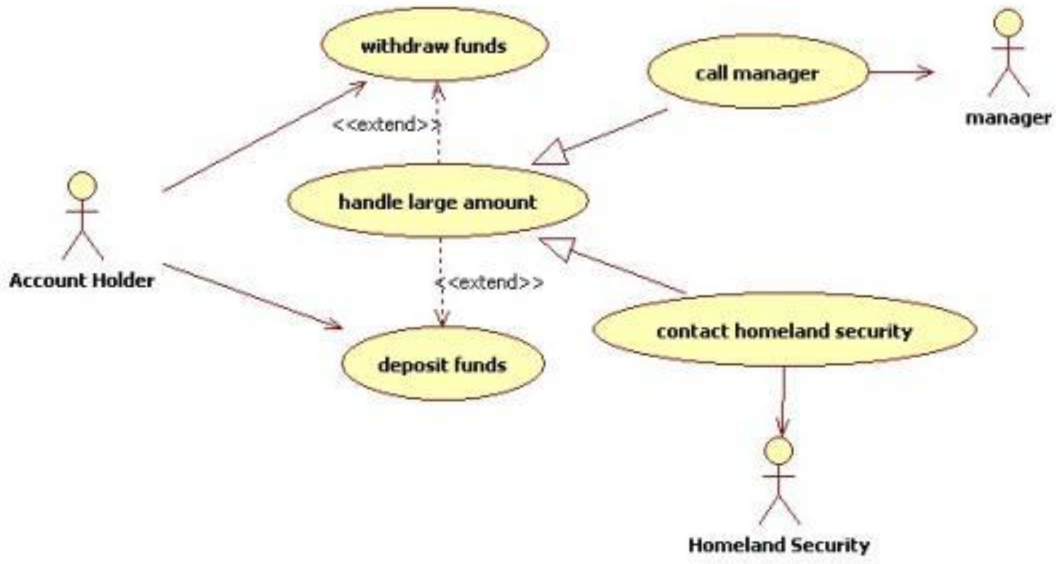
```
class HandleLargeAmount implements UseCase {
�� public void execute(Model context) throws Exception {
����� // check holders zip code
�� }
}
```

Here is how we plug it in to a deposit funds controller:

```
DepositFunds depositController = new DepositFunds();
HandleLargeAmount handler = new HandleLargeAmount();
depositController.setLargeAmountHandler(handler);
```

Generalization

A use case can have several specializations. We can show this using the generalization arrow:

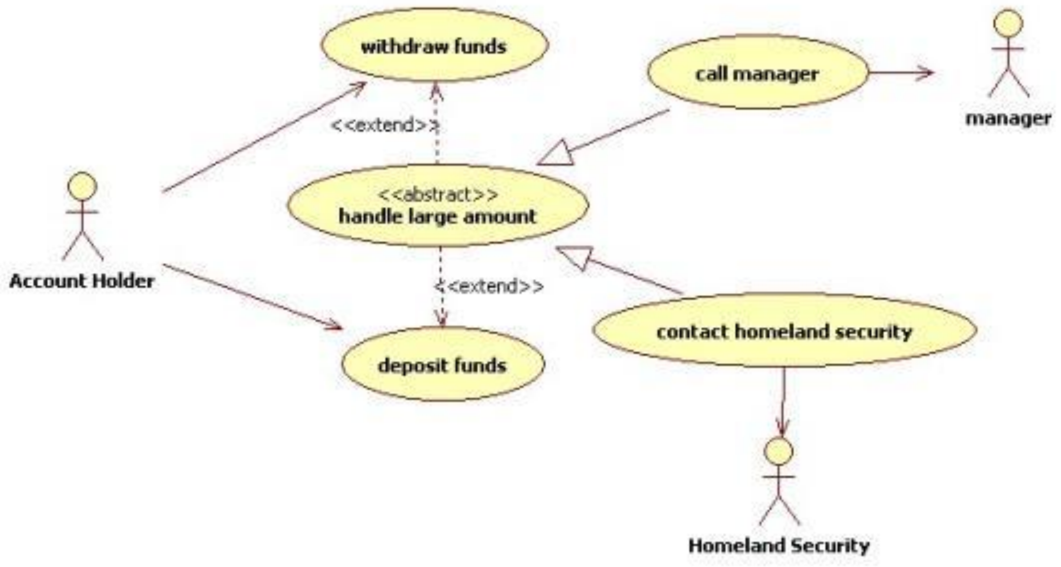


This can be implemented by extending use case controllers that redefine the execute method:

```
class ContactHomelandSecurity extends HandleLargeAmount {
�� private HomelandSecurityServer server;
�� public void execute(Model context) throws Exception {
����� server.report();
�� }
}
```

Abstract Use Cases

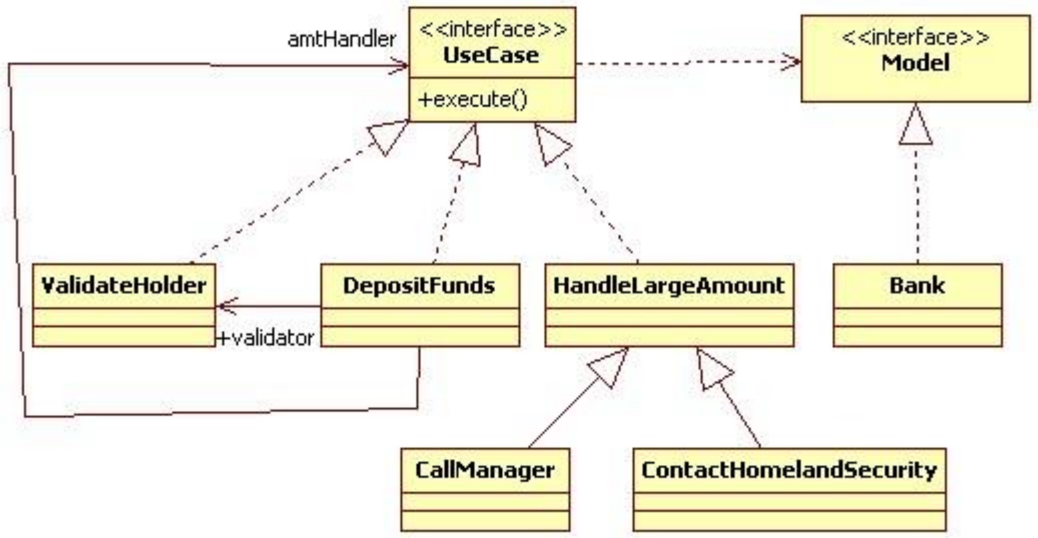
Like classes, use cases can be abstract:



This can be implemented as an abstract use case controller:

```
abstract class HandleLargeAmount implements UseCase {
�� abstract public void execute(Model context) throws Exception;
}
```

Design Model for Use Case Controllers



OBJECT ORIENTED SOFTWARE ENGINEERING

UNIT-2

Architecture

System development is model building

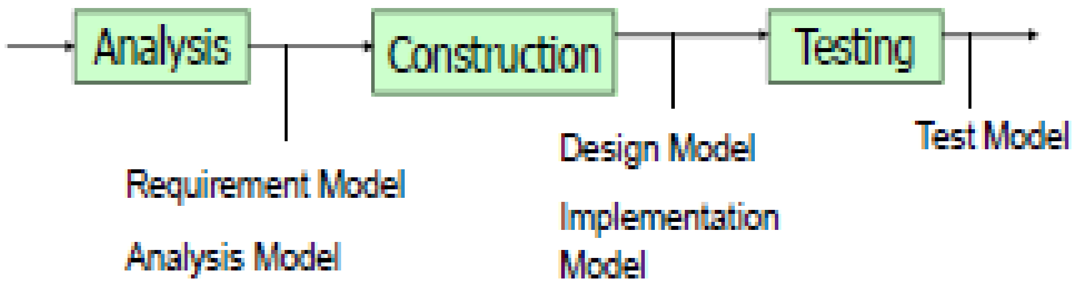
The work that occurs when we develop computer support to aid an organization. System development is model building. It starts when a requirement of system identified and Specification can be used for contract and to plan and control the development process. It is a complex process handle poorly. OOSE can used from start to end of system life cycle. It is based on three technologies:

- Object oriented programming
- Conceptual modeling
- Block design

It consists of five different models:

- ***The requirement model***
  Aims to capture the functional requirements
- ***Analysis model***
  Give the system a robust and changeable object structure
- ***Design model***
  Adopt and refine the object structure to the current implementation environment
- ***Implementation model***
  Implement the system
- ***Test model***
  Verify the system

Seamless transition between the models is important. The method layer define the transformation rules. For maintainability traceability is very important between the models.



Models are tightly coupled to the architecture, and aim is to find concepts which are simple to learn and use. It simplify our understanding of the system and provide us with a changeable model of the system. These models are sufficiently powerful to express the information which is required to model the system. These models are sufficiently defined that different people can discuss the system in terms of these concepts without being misunderstood.

<u>**model architecture**</u>

Software architecture involves the high level structure of software system abstraction, by using decomposition and composition, with architectural style and quality attributes. A software architecture design must conform to the major functionality and performance requirements of the system, as well as satisfy the non-functional requirements such as reliability, scalability, portability, and availability.

A software architecture must describe its group of components, their connections, interactions among them and deployment configuration of all components.

A software architecture can be defined in many ways −

- **UML (Unified Modeling Language)** − UML is one of object-oriented solutions used in software modeling and design.
- **Architecture View Model (4+1 view model)** − Architecture view model represents the functional and non-functional requirements of software application.
- **ADL (Architecture Description Language)** − ADL defines the software architecture formally and semantically.

UML

UML stands for Unified Modeling Language. It is a pictorial language used to make software blueprints. UML was created by Object Management Group (OMG). The UML 1.0 specification draft was proposed to the OMG in January 1997. It serves as a standard for software requirement analysis and design documents which are the basis for developing a software.

UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document a software system. Although UML is generally used to model software system, it is not limited within this boundary. It is also used to model non software systems such as process flows in a manufacturing unit.

The elements are like components which can be associated in different ways to make a complete UML picture, which is known as a **diagram**. So, it is very important to understand the different diagrams to implement the knowledge in real-life systems. We have two broad categories of diagrams and they are further divided into sub-categories i.e. **Structural Diagrams** and **Behavioral Diagrams**.

Structural Diagrams

Structural diagrams represent the static aspects of a system. These static aspects represent those parts of a diagram which forms the main structure and is therefore stable.

These static parts are represented by classes, interfaces, objects, components and nodes. Structural diagrams can be sub-divided as follows −

- Class diagram
- Object diagram
- Component diagram
- Deployment diagram
- Package diagram
- Composite structure

The following table provides a brief description of these diagrams −

Sr.No.	Diagram & Description
1	**Class** Represents the object orientation of a system. Shows how classes are statically related.
2	**Object** Represents a set of objects and their relationships at runtime and also represent the static view of the system.
3	**Component** Describes all the components, their interrelationship, interactions and interface of the system.
4	**Composite structure** Describes inner structure of component including all classes, interfaces of the component, etc.
5	**Package** Describes the package structure and organization. Covers classes in the package and packages within another package.

6	**Deployment** Deployment diagrams are a set of nodes and their relationships. These nodes are physical entities where the components are deployed.

Behavioral Diagrams

Behavioral diagrams basically capture the dynamic aspect of a system. Dynamic aspects are basically the changing/moving parts of a system. UML has the following types of behavioral diagrams −

- Use case diagram
- Sequence diagram
- Communication diagram
- State chart diagram
- Activity diagram
- Interaction overview
- Time sequence diagram

The following table provides a brief description of these diagram −

Sr.No.	Diagram & Description
1	**Use case** Describes the relationships among the functionalities and their internal/external controllers. These controllers are known as actors.
2	**Activity** Describes the flow of control in a system. It consists of activities and links. The flow can be sequential, concurrent, or branched.
3	**State Machine/state chart** Represents the event driven state change of a system. It basically describes the state change of a class, interface, etc. Used to visualize the reaction of a system by internal/external factors.
4	**Sequence** Visualizes the sequence of calls in a system to perform a specific functionality.

5	**Interaction Overview** Combines activity and sequence diagrams to provide a control flow overview of system and business process.
6	**Communication** Same as sequence diagram, except that it focuses on the object's role. Each communication is associated with a sequence order, number plus the past messages.
7	**Time Sequenced** Describes the changes by messages in state, condition and events.

Architecture View Model

A model is a complete, basic, and simplified description of software architecture which is composed of multiple views from a particular perspective or viewpoint.

A view is a representation of an entire system from the perspective of a related set of concerns. It is used to describe the system from the viewpoint of different stakeholders such as end-users, developers, project managers, and testers.

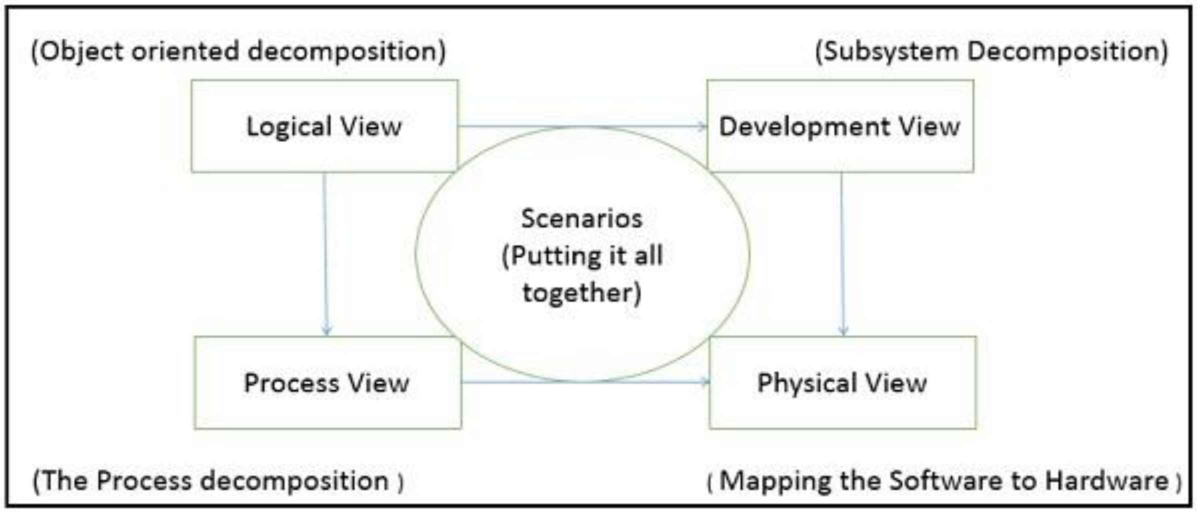
4+1 View Model

The 4+1 View Model was designed by Philippe Kruchten to describe the architecture of a software–intensive system based on the use of multiple and concurrent views. It is a **multiple view** model that addresses different features and concerns of the system. It standardizes the software design documents and makes the design easy to understand by all stakeholders.

It is an architecture verification method for studying and documenting software architecture design and covers all the aspects of software architecture for all stakeholders. It provides four essential views −

- **The logical view or conceptual view** − It describes the object model of the design.
- **The process view** − It describes the activities of the system, captures the concurrency and synchronization aspects of the design.
- **The physical view** − It describes the mapping of software onto hardware and reflects its distributed aspect.
- **The development view** − It describes the static organization or structure of the software in its development of environment.

This view model can be extended by adding one more view called **scenario view** or **use case view** for end-users or customers of software systems. It is coherent with other four views and are utilized to illustrate the architecture serving as “plus one” view, (4+1) view model. The following figure describes the software architecture using five concurrent views (4+1) model.


(Object oriented decomposition)
(Subsystem Decomposition)
Logical View
Development View
Scenarios (Putting it all together)
Process View
Physical View
(The Process decomposition)
(Mapping the Software to Hardware)

Why is it called 4+1 instead of 5?

The **use case view** has a special significance as it details the high level requirement of a system while other views details — how those requirements are realized. When all other four views are completed, it’s effectively redundant. However, all other views would not be possible without it. The following image and table shows the 4+1 view in detail –

	Logical	**Process**	**Development**	**Physical**	**Scenario**
Description	Shows the component (Object) of system as well as their interaction	Shows the processes / Workflow rules of system and how those processes communicate, focuses on dynamic view of system	Gives building block views of system and describe static organization of the system modules	Shows the installation, configuration and deployment of software application	Shows the design is complete by performing validation and illustration
Viewer / Stake holder	End-User, Analysts and Designer	Integrators & developers	Programmer and software project managers	System engineer, operators, system administrators and system installers	All the views of their views and evaluators
Consider	Functional requirements	Non Functional Requirements	Software Module organization (Software management reuse, constraint of tools)	Nonfunctional requirement regarding to underlying hardware	System Consistency and validity

UML – Diagram	Class, State, Object, sequence, Communication Diagram	Activity Diagram	Component, Package diagram	Deployment diagram	Use case diagram

Architecture Description Languages (ADLs)

An ADL is a language that provides syntax and semantics for defining a software architecture. It is a notation specification which provides features for modeling a software system's conceptual architecture, distinguished from the system's implementation.

ADLs must support the architecture components, their connections, interfaces, and configurations which are the building block of architecture description. It is a form of expression for use in architecture descriptions and provides the ability to decompose components, combine the components, and define the interfaces of components.

An architecture description language is a formal specification language, which describes the software features such as processes, threads, data, and sub-programs as well as hardware component such as processors, devices, buses, and memory.

It is hard to classify or differentiate an ADL and a programming language or a modeling language. However, there are following requirements for a language to be classified as an ADL −

- It should be appropriate for communicating the architecture to all concerned parties.
- It should be suitable for tasks of architecture creation, refinement, and validation.
- It should provide a basis for further implementation, so it must be able to add information to the ADL specification to enable the final system specification to be derived from the ADL.
- It should have the ability to represent most of the common architectural styles.
- It should support analytical capabilities or provide quick generating prototype implementations.

requirement model

Requirements modeling in software engineering identifies the **requirements** that a software application or system must meet in order to solve the business problem. **Requirements** are divided into functional (what the system will have to do) and non-functional (constraints within which the system will have to perform).

analysis model

An object model describing the realization of use cases, and which serves as an abstraction of the Artifact: Design Model. The Analysis Model contains the results of use case analysis, instances of the Artifact: Analysis Class. The Analysis Model is an optional artifact.

design model

The **design model** is an object **model** describing the realization of use cases, and serves as an abstraction of the implementation **model** and its source code. The **design model** is used as essential input to activities in implementation and test.

implementation model

An **implementation model** is a representation of how a system (application, service, interface etc.) actually works. It's often described with system diagrams and pseudo code, and later translated into real code.

Analysis

System development based on user requirement

1. Developers must be trained not to add any features or capabilities beyond those stated in the approved requirements. Industry experience is that developers add capabilities, increasing the scope of the product and adding to the effort required throughout the system's lifecycle.

2. Methods such as requirements workshops (conducted as needed), storyboards, requirements reviews, idea reduction, role playing, and prototyping should be used. See Dean Leffingwell, *Managing Software Requirements: A Unified Approach*, for a lot of experience and suggestions.

3. "What if" questions should be asked, focusing on the boundary conditions, exceptions, and unusual events.

4. A careful manual review and analysis of the complete set of requirements, supported by the user and utilizing appropriate automated tools, will be needed to ensure consistency in the requirements set.

5. Developers, customers, and other stakeholders need to rank the individual requirements in terms of their importance to the customer. This technique helps to manage the project scope. Requirements should also be ranked in terms of their stability. This is only a beginning to defining the releases of the product. Data dependencies must be considered as well. See Bill Wiley, *Essential System Requirements: A Practical Guide to Event-Driven Methods*, Chapters 8 and 9 for discussion of data/process interaction and transition to physical design.

6. One of the most difficult requirements challenges is writing the requirements detailed enough to be well understood without over-constraining the system and annoying users with details intended to remove ambiguity. Here are some guidelines:

   - Use natural language whenever possible.
   - Use pictures and diagrams to further illustrate the intent.
   - When in doubt, ask for more information.
   - Augment specifications with more formal methods when it is critical not to be misunderstood (life-and-death situations or grave consequences of erroneous behavior).
   - Train people to recognize the problem and the solution.
   - Use diagrams and structured pseudo-code to describe complex technical specifications.

7. We *know* that the requirements will change for several reasons:

   1. External factors:
      a. Change in the problem (the business) we are attempting to solve.
      b. Users change their minds.
      c. The external environment has changed, creating new constraints and/or opportunities (for example, the availability of the Internet and the World Wide Web).
      d. The very existence of the new system causes the requirements for the system to change!

   2. Internal factors:

      a. Failure to discover the real requirements during the initial requirements-gathering effort.
      b. Failure to create a practical process to help manage changes.

   - Requirements leakage:
      a. Direct user/customer requests to programmers.

b. Functionality inserted by programmers "because it's a good thing."

Change must be managed. Here are some guidelines:

- Recognize that change is inevitable and provide methods to deal with it.
- Baseline the requirements.
- Establish a single channel to control change (such as the joint team).
- Use a change-control system to capture changes. Keep requirements under vigorous configuration management (CM).
- Manage changes hierarchically so that the downward ripple effect of a change will be highlighted by the traceability in the requirements tool.
- Industry data show that a change of 30% in the requirements results in *doubling* of the costs of the project. The joint team is critical to provide joint customer and contractor responsibility for the requirements and for the impacts of any approved changes.

Use case model

A use-case model is a model of how different types of users interact with the system to solve a problem. As such, it describes the goals of the users, the interactions between the users and the system, and the required behavior of the system in satisfying these goals.

A use-case model consists of a number of model elements. The most important model elements are: use cases, actors and the relationships between them.

A use-case diagram is used to graphically depict a subset of the model to simplify communications. There will typically be several use-case diagrams associated with a given model, each showing a subset of the model elements relevant for a particular purpose. The same model element may be shown on several use-case diagrams, but each instance must be consistent. If tools are used to maintain the use-case model, this consistency constraint is automated so that any changes to the model element (changing the name for example) will be automatically reflected on every use-case diagram that shows that element.

The use-case model may contain packages that are used to structure the model to simplify analysis, communications, navigation, development, maintenance and planning.

Much of the use-case model is in fact textual, with the text captured in the Use-Case Specifications that are associated with each use-case model element. These specifications describe the flow of events of the use case.

The use-case model serves as a unifying thread throughout system development. It is used as the primary specification of the functional requirements for the system, as the basis for analysis and design, as an input to iteration planning, as the basis of defining test cases and as the basis for user documentation.

interface descriptions

An **interface description language** or **interface definition language** (**IDL**), is a specification language used to describe a software component's application programming interface (API). IDLs describe an interface in a language-independent way, enabling communication between software components that do not share one language, for example, between those written in C++ and those written in Java.

IDLs are commonly used in remote procedure call software. In these cases the machines at either end of the *link* may be using different operating systems and computer languages. IDLs offer a bridge between the two different systems.

Software systems based on IDLs include Sun's ONC RPC, The Open Group's Distributed Computing Environment, IBM's System Object Model, the Object Management Group's CORBA (which implements OMG IDL, an IDL based on DCE/RPC) and Data Distribution Service, Mozilla's XPCOM, Microsoft's Microsoft RPC (which evolved into COM and DCOM), Facebook's Thrift and WSDL for Web services.

Problem domain objects

In a nutshell anything substantial you find around a **problem domain** is considered a **domain object**. In your case you have the receptionist, the SMS, the reservation and the customer. Now what you are doing is to classify these **objects** (make classes). And that's called **problem domain** analysis.

entity objects

Entity objects are classes that encapsulate the business model, including rules, data, relationships, and persistence behavior, for items that are used in your business application. For example, **entity objects** can represent. the logical structure of the business, such as product lines, departments, sales, and regions.

control objects

Entities are **objects** representing system data: Customer, Transaction, Cart, etc. Boundaries are **objects** that interface with system actors: user interfaces, gateways, proxies, etc. Controllers are **objects** that mediate between boundaries and entities. They orchestrate the execution of commands coming from the boundary.

Code Design Improvement

Refactoring

Code refactoring is the process of restructuring existing computer code—changing the *factoring*—without changing its external behavior. Refactoring is intended to improve *nonfunctional* attributes of the software. Advantages include improved code readability and reduced complexity; these can improve source-code maintainability and create a more expressive internal architecture or object model to improve extensibility.

Typically, refactoring applies a series of standardised basic *micro-refactorings*, each of which is (usually) a tiny change in a computer program's source code that either preserves the behaviour of the software, or at least does not modify its conformance to functional requirements.
Many development environments provide automated support for performing the mechanical aspects of these basic refactorings. If done well, code refactoring may help software developers discover and fix hidden or dormant bugs or vulnerabilities in the system by simplifying the underlying logic and eliminating unnecessary levels of complexity. If done poorly it may fail the requirement that external functionality not be changed, introduce new bugs, or both.

By continuously improving the design of code, we make it easier and easier to work with. This is in sharp contrast to what typically happens: little refactoring and a great deal of attention paid to expediently adding new features. If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code.

Anti patterns

An **anti-pattern** is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive.[1][2] The term, coined in 1995 by Andrew Koenig,[3] was inspired by a book, *Design Patterns*, which highlights a number of design patterns in software development that its authors considered to be highly reliable and effective.

The term was popularized three years later by the book *AntiPatterns*, which extended its use beyond the field of software design to refer informally to any commonly reinvented but bad solution to a problem. Examples include analysis paralysis, cargo cult programming, death march, groupthink and vendor lock-in.

Visitor Patterns

In object-oriented programming and software engineering, the **visitor** design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying the structures. It is one way to follow the open/closed principle.

In essence, the visitor allows adding new virtual functions to a family of classes, without modifying the classes. Instead, a visitor class is created that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch.

OBJECT ORIENTED SOFTWARE ENGINEERING

UNIT-3

Construction

Software construction is a software engineering discipline. It is the detailed creation of working meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging. It is linked to all the other software engineering disciplines, most strongly to software design and software testing.

The design model

A **design model** in Software Engineering is an object-based picture or pictures that represent the use cases for a system. Or to put it another way, it is the means to describe a system's implementation and source code in a diagrammatic fashion. This type of representation has a couple of advantages. First, it is a simpler representation than words alone. Second, a group of people can look at these simple diagrams and quickly get the general idea behind a system. In the end, it boils down to the old adage, 'a picture is worth a thousand words.'

Design Model Dimensions

- The design model can be viewed in two different dimensions.
- (Horizontally) The process dimension
- It indicates the evolution of the parts of the design model as each design task is executed.
- (Vertically) The abstraction dimension
- It represents the level of detail as each element of the analysis model is transformed into the design model and then iteratively refined.
- The elements of the design model use many of the same UML diagrams that were used in the analysis model.
- The difference is that these diagrams are
- Refined and elaborated as part of design;
- More implementation-specific detail is provided,
- Architectural structure and style, components that reside within the architecture,
- Interfaces between the components and with the outside world are all emphasized.

Block Design

In combinatorial mathematics, a **block design** is a set together with a family of subsets (repeated subsets are allowed at times) whose members are chosen to satisfy some set of properties that are deemed useful for a particular application. These applications come from many areas, including experimental design, finite geometry, physical chemistry, software testing, cryptography, and algebraic geometry. Many variations have been examined, but the most intensely studied are the **balanced incomplete block designs** (BIBDs or 2-designs) which historically were related to statistical issues in the design of experiments.[1][2]

A block design in which all the blocks have the same size is called *uniform*. The designs discussed in this article are all uniform. Pairwise balanced designs (PBDs) are examples of block designs that are not necessarily uniform.

Working with construction

Software construction fundamentals

Minimizing complexity

The need to reduce complexity is mainly driven by limited ability of most people to hold complex structures and information in their working memories. Reduced complexity is achieved through emphasizing the creation of code that is simple and readable rather than clever.
Minimizing complexity is accomplished through making use of standards, and through numerous specific techniques in coding. It is also supported by the construction-focused quality techniques.

Anticipating change

Anticipating change helps software engineers build extensible software, which means they can enhance a software product without disrupting the underlying structure.[2] Research over 25 years showed that the cost of rework can be 10 to 100 times (5 to 10 times for smaller projects) more expensive than getting the requirements right the first time. Given that 25% of the requirements change during development on average project, the need to reduce the cost of rework elucidates the need for anticipating change.

Constructing for verification

Constructing for verification means building software in such a way that faults can be ferreted out readily by the software engineers writing the software, as well as during independent testing and operational activities. Specific techniques that support constructing for verification include following coding standards to support code reviews, unit testing, organizing code to support automated testing, and restricted use of complex or hard-to-understand language structures, among others.

Reuse

Systematic reuse can enable significant software productivity, quality, and cost improvements. Reuse has two closely related facets:

- Construction for reuse: Create reusable software assets.
- Construction with reuse: Reuse software assets in the construction of a new solution.

Standards in construction

Standards, whether external (created by international organizations) or internal (created at the corporate level), that directly affect construction issues include:

- Communication methods: Such as standards for document formats and contents.
- Programming languages
- Coding standards
- Platforms
- Tools: Such as diagrammatic standards for notations like UML.

Managing construction

Construction model

Numerous models have been created to develop software, some of which emphasize construction more than others. Some models are more linear from the construction point of view, such as the Waterfall and staged-delivery life cycle models. These models treat construction as an activity which occurs only after significant prerequisite work has been completed—including detailed requirements work, extensive design work, and detailed planning. Other models are more iterative, such as evolutionary prototyping, Extreme Programming, and Scrum. These approaches tend to treat construction as an activity that occurs concurrently with other software development activities, including requirements, design, and planning, or overlaps them.

Construction planning

The choice of construction method is a key aspect of the construction planning activity. The choice of construction method affects the extent to which construction prerequisites (e.g. Requirements analysis, Software design, .. etc.) are performed, the order in which they are performed, and the degree to which they are expected to be completed before construction work begins. Construction planning also defines the order in which components are created and integrated, the software quality management processes, the allocation of task assignments to specific software engineers, and the other tasks, according to the chosen method.

Construction measurement

Numerous construction activities and artifacts can be measured, including code developed, code modified, code reused, code destroyed, code complexity, code inspection statistics, fault-fix and fault-find rates, effort, and scheduling. These measurements can be useful for purposes of managing construction, ensuring quality during construction, improving the construction process, as well as for other reasons.

Practical considerations

Software construction is driven by many practical considerations:

Construction design

In order to account for the unanticipated gaps in the software design, during software construction some design modifications must be made on a smaller or larger scale to flesh out details of the software design.

Low Fan-out is one of the design characteristics found to be beneficial by researchers. Information hiding proved to be a useful design technique in large programs that made them easier to modify by a factor of 4.

Construction languages

Construction languages include all forms of communication by which a human can specify an executable problem solution to a computer. They include configuration languages, toolkit languages, and programming languages:

- Configuration languages are languages in which software engineers choose from a limited set of predefined options to create new or custom software installations.
- Toolkit languages are used to build applications out of toolkits and are more complex than configuration languages.
- Scripting languages are kinds of application programming languages that supports scripts which are often interpreted rather than compiled.
- Programming languages are the most flexible type of construction languages which use three general kinds of notation:
    - Linguistic notations which are distinguished in particular by the use of word-like strings of text to represent complex software constructions, and the combination of such word-like strings into patterns that have a sentence-like syntax.
    - Formal notations which rely less on intuitive, everyday meanings of words and text strings and more on definitions backed up by precise, unambiguous, and formal (or mathematical) definitions.
    - Visual notations which rely much less on the text-oriented notations of both linguistic and formal construction, and instead rely on direct visual interpretation and placement of visual entities that represent the underlying software.

Programmers working in a language they have used for three years or more are about 30 percent more productive than programmers with equivalent experience who are new to a language. High-level languages such as C++, Java, Smalltalk, and Visual Basic yield 5 to 15 times better productivity, reliability, simplicity, and comprehensibility than low-level languages such as assembly and C. Equivalent code has been shown to need fewer lines to be implemented in high level languages than in lower level languages.

Coding

Main article: Computer programming

The following considerations apply to the software construction coding activity:

- Techniques for creating understandable source code, including naming and source code layout. One study showed that the effort required to debug a program is minimized when the variables' names are between 10 and 16 characters.
- Use of classes, enumerated types, variables, named constants, and other similar entities:
    - A study done by NASA showed that the putting the code into well-factored classes can double the code reusability compared to the code developed using functional design.
    - One experiment showed that designs which access arrays sequentially, rather than randomly, result in fewer variables and fewer variable references.
- Use of control structures:
    - One experiment found that loops-with-exit are more comprehensible than other kinds of loops.
    - Regarding the level of nesting in loops and conditionals, studies have shown that programmers have difficulty comprehending more than three levels of nesting.
    - Control flow complexity has been shown to correlate with low reliability and frequent errors.
- Handling of error conditions—both planned errors and exceptions (input of bad data, for example)
- Prevention of code-level security breaches (buffer overruns or array index overflows, for example)
- Resource usage via use of exclusion mechanisms and discipline in accessing serially reusable resources (including threads or database locks)
- Source code organization (into statements and routines):

    - Highly cohesive routines proved to be less error prone than routines with lower cohesion. A study of 450 routines found that 50 percent of the highly cohesive routines were fault free compared to only 18 percent of routines with low cohesion. Another study of a different 450 routines found that routines with the highest coupling-to-cohesion ratios had 7 times as many errors as those with the lowest coupling-to-cohesion ratios and were 20 times as costly to fix.
    - Although studies showed inconclusive results regarding the correlation between routine sizes and the rate of errors in them, but one study found that routines with fewer than 143 lines of code were 2.4 times less expensive to fix than larger routines. Another study showed that the code needed to be changed least when routines averaged 100 to 150 lines of code. Another study found that structural complexity and amount of data in a routine were correlated with errors regardless of its size.
    - Interfaces between routines are some of the most error-prone areas of a program. One study showed that 39 percent of all errors were errors in communication between routines.
    - Unused parameters are correlated with an increased error rate. In one study, only 17 to 29 percent of routines with more than one unreferenced variable had no errors, compared to 46 percent in routines with no unused variables.
    - The number of parameters of a routine should be 7 at maximum as research has found that people generally cannot keep track of more than about seven chunks of information at once.
- Source code organization (into classes, packages, or other structures). When considering containment, the maximum number of data members in a class shouldn't exceed 7±2. Research has shown that this number is the number of discrete items a person can remember while performing other tasks. When considering inheritance, the number of levels in the inheritance tree should be limited. Deep inheritance trees have been found to be significantly associated with increased fault rates. When considering the number of routines in a class, it should be kept as small as possible. A study on C++ programs has found an association between the number of routines and the number of faults.[10]
- Code documentation
- Code tuning

Construction testing

The purpose of construction testing is to reduce the gap between the time at which faults are inserted into the code and the time those faults are detected. In some cases, construction testing is performed after code has been written. In test-first programming, test cases are created before code is written. Construction involves two forms of testing, which are often performed by the software engineer who wrote the code:[1]

- Unit testing
- Integration testing

Reuse

Implementing software reuse entails more than creating and using libraries of assets. It requires formalizing the practice of reuse by integrating reuse processes and activities into the software life cycle. The tasks related to reuse in software construction during coding and testing are:[1]

- The selection of the reusable units, databases, test procedures, or test data.
- The evaluation of code or test re-usability.
- The reporting of reuse information on new code, test procedures, or test data.

Construction quality

The primary techniques used to ensure the quality of code as it is constructed include:

- Unit testing and integration testing. One study found that the average defect detection rates of unit testing and integration testing are 30% and 35% respectively.[16]
- Test-first development
- Use of assertions and defensive programming
- Debugging
- Inspections. One study found that the average defect detection rate of formal code inspections is 60%. Regarding the cost of finding defects, a study found that code reading detected 80% more faults per hour than testing. Another study shown that it costs six times more to detect design defects by using testing than by using inspections. A study by IBM showed that only 3.5 hours where needed to find a defect through code inspections versus 15–25 hours through testing. Microsoft has found that it takes 3 hours to find and fix a defect by using code inspections and 12 hours to find and fix a defect by using testing. In a 700 thousand lines program, it was reported that code reviews were several times as cost-effective as testing.[16] Studies found that inspections result in 20% - 30% fewer defects per 1000 lines of code than less formal review practices and that they increase productivity by about 20%. Formal inspections will usually take 10% - 15% of the project budget and will reduce overall project cost. Researchers found that having more than 2 - 3 reviewers on a formal inspection doesn't increase the number of defects found, although the results seem to vary depending on the kind of material being inspected.
- Technical reviews. One study found that the average defect detection rates of informal code reviews and desk checking are 25% and 40% respectively.[16] Walkthroughs were found to have defect detection rate of 20% - 40%, but were found also to be expensive specially when project pressures increase. Code reading was found by NASA to detect 3.3 defects per hour of effort versus 1.8 defects per hour for testing. It also finds 20% - 60% more errors over the life of the project than different kinds of testing. A study of 13 reviews about review meetings, found that 90% of the defects were found in preparation for the review meeting while only around 10% were found during the meeting.
- Static analysis (IEEE1028)

Studies have shown that a combination of these techniques need to be used to achieve high defect detection rate. Other studies showed that different people tend to find different defects. One study found that the Extreme Programming practices of pair programming, desk checking, unit testing, integration testing, and regression testing can achieve a 90% defect detection rate. An experiment involving experienced programmers found that on average they were able to find 5 errors (9 at best) out of 15 errors by testing.

80% of the errors tend to be concentrated in 20% of the project's classes and routines. 50% of the errors are found in 5% of the project's classes. IBM was able to reduce the customer reported defects by a factor of ten to one and to reduce their maintenance budget by 45% in its IMS system by repairing or rewriting only 31 out of 425 classes. Around 20% of a project's routines contribute to 80% of the development costs. A classic study by IBM found that few error-prone routines of OS/360 were the most expensive entities. They had around 50 defects per 1000 lines of code and fixing them costs 10 times what it took to develop the whole system.

Integration

A key activity during construction is the integration of separately constructed routines, classes, components, and subsystems. In addition, a particular software system may need to be integrated with other software or hardware systems. Concerns related to construction integration include planning the sequence in which components will be integrated, creating scaffolding to support interim versions of the software, determining the degree of testing and quality work performed on components before they are integrated, and determining points in the project at which interim versions of the software are tested.[1]

Testing

Software testing can be stated as the process of verifying and validating that a software or application is bug free, meets the technical requirements as guided by it's design and development and meets the user requirements effectively and efficiently with handling all the exceptional and boundary cases.

The process of software testing aims not only at finding faults in the existing software but also at finding measures to improve the software in terms of efficiency, accuracy and usability. It mainly aims at measuring specification, functionality and performance of a software program or application.

Software testing can be divided into two steps:
1. **Verification:** it refers to the set of tasks that ensure that software correctly implements a specific function.
2. **Validation:** it refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
Verification: "Are we building the product right?"
Validation: "Are we building the right product?"

What are different types of software testing?
Software Testing can be broadly classified into two types:

1. **Manual Testing:** Manual testing includes testing a software manually, i.e., without using any automated tool or any script. In this type, the tester takes over the role of an end-user and tests the software to identify any unexpected behavior or bug. There are different stages for manual testing such as unit testing, integration testing, system testing, and user acceptance testing. Testers use test plans, test cases, or test scenarios to test a software to ensure the completeness of testing. Manual testing also includes exploratory testing, as testers explore the software to identify errors in it.

2. **Automation Testing:** Automation testing, which is also known as Test Automation, is when the tester writes scripts and uses another software to test the product. This process involves automation of a manual process. Automation Testing is used to re-run the test scenarios that were performed manually, quickly, and repeatedly.
Apart from regression testing, automation testing is also used to test the application from load, performance, and stress point of view. It increases the test coverage, improves accuracy, and saves time and money in comparison to manual testing.

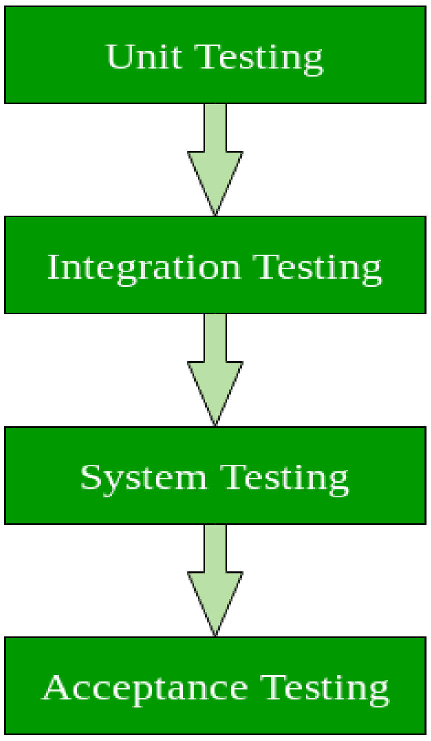
What are different techniques of Software Testing?
Software techniques can be majorly classified into two categories:

1. **Black Box Testing:** The technique of testing in which the tester doesn't have access to the source code of the software and is conducted at the software interface without concerning with the internal logical structure of the software is known as black box testing.
2. **White-Box Testing:** The technique of testing in which the tester is aware of the internal workings of the product, have access to it's source code and is conducted by making sure that all internal operations are performed according to the specifications is known as white box testing.

BLACK BOX TESTING	WHITE BOX TESTING
Internal workings of an application are not required.	Knowledge of the internal workings is must.
Also known as closed box/data driven testing.	Also knwon as clear box/structural testing.
End users, testers and developers.	Normally done by testers and developers.
THis can only be done by trial and error method.	Data domains and internal boundaries can be better tested.

What are different levels of software testing?
Software level testing can be majorly classified into 4 levels:

1. **Unit Testing:** A level of the software testing process where individual units/components of a software/system are tested. The purpose is to validate that each unit of the software performs as designed.
2. **Integration Testing:** A level of the software testing process where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.
3. **System Testing:** A level of the software testing process where a complete, integrated system/software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.
4. **Acceptance Testing:** A level of the software testing process where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.



Object Oriented testing process

Software typically undergoes many levels of testing, from unit testing to system or acceptance testing. Typically, in-unit testing, small “units”, or modules of the software, are tested separately with focus on testing the code of that module. In higher, order testing (e.g, acceptance testing), the entire system (or a subsystem) is tested with the focus on testing the functionality or external behavior of the system.
As information systems are becoming more complex, the object-oriented paradigm is gaining popularity because of its benefits in analysis, design, and coding. Conventional testing methods cannot be applied for testing classes because of problems involved in testing classes, abstract classes, inheritance, dynamic binding, message, passing, polymorphism, concurrency, etc.
Testing classes is a fundamentally different problem than testing functions. A function (or a procedure) has a clearly defined input-output behavior, while a class does not have an input-output behavior specification. We can test a method of a class using approaches for testing functions, but we cannot test the class using these
approaches.

According to Davis the dependencies occurring in conventional systems are:

- Data dependencies between variables
- Calling dependencies between modules
- Functional dependencies between a module and the variable it computes
- Definitional dependencies between a variable and its types.

But in Object-Oriented systems there are following additional dependencies:

- Class to class dependencies
- Class to method dependencies
- Class to message dependencies
- Class to variable dependencies
- Method to variable dependencies
- Method to message dependencies
- Method to method dependencies

Issues in Testing Classes:
Additional testing techniques are, therefore, required to test these dependencies. Another issue of interest is that it is not possible to test the class dynamically, only its instances i.e, objects can be tested. Similarly, the concept of inheritance opens various issues e.g., if changes are made to a parent class or superclass, in a larger system of a class it will be difficult to test subclasses individually and isolate the error to one class.

In object-oriented programs, control flow is characterized by message passing among objects, and the control flow switches from one object to another by inter-object communication. Consequently, there is no control flow within a class like functions. This lack of sequential control flow within a class requires different approaches for testing. Furthermore, in a function, arguments passed to the function with global data determine the path of execution within the procedure. But, in an object, the state associated with the object also influences the path of execution, and methods of a class can communicate among themselves through this state because this state is persistent across invocations of methods. Hence, for testing objects, the state of an object has to play an important role.

Techniques of object-oriented testing are as follows:

1. **Fault Based Testing:**
   This type of checking permits for coming up with test cases supported the consumer specification or the code or both. It tries to identify possible faults (areas of design or code that may lead to errors.). For all of these faults, a test case is developed to "flush" the errors out. These tests also force each time of code to be executed.
   This method of testing does not find all types of errors. However, incorrect specification and interface errors can be missed. These types of errors can be uncovered by function testing in the traditional testing model. In the object-oriented model, interaction errors can be uncovered by scenario-based testing. This form of Object oriented-testing can only test against the client's specifications, so interface errors are still missed.
2. **Class Testing Based on Method Testing:**
   This approach is the simplest approach to test classes. Each method of the class performs a well defined cohesive function and can, therefore, be related to unit testing of the traditional testing techniques. Therefore all the methods of a class can be involved at least once to test the class.
3. **Random Testing:**
   It is supported by developing a random test sequence that tries the minimum variety of operations typical to the behavior of the categories
4. **Partition Testing:**
   This methodology categorizes the inputs and outputs of a category so as to check them severely. This minimizes the number of cases that have to be designed.
5. **Scenario-based Testing:**
   It primarily involves capturing the user actions then stimulating them to similar actions throughout the test.
   These tests tend to search out interaction form of error.

<u>testing of analysis and design model</u>

The systematic testing of analysis and design models is a labor-intensive exercise that is highly efficient. The technique can be made more efficient by a careful selection of use cases to serve as test cases. Depending upon the selection technique, the faults that would have the most critical implications for the system or those that would occur most frequently can be targeted. Defects identified at this level affect a wider scope of the system and can be removed with less effort than defects found at the more detailed level of code testing. The activities I described here work closely with the typical development techniques used by developers following an object-oriented method. By integrating these techniques into the development process, the process becomes self-validating in that one development activity identifies defects in the products of previous development tasks. Testing becomes a continuous process that guides development rather than a judgmental step at the end of the development process. Several of these techniques can be automated, further reducing the effort required. The result is an improved system which ultimately will be of higher quality and in many cases at a lower cost due to early detection.

Testing of classes

Object oriented software is based on concept of classes and objects. Therefore, smallest unit to be tested in a class.
There are some issues in class testing.

1. A class can't be tested directly first one need to create instances of class, as object then test it. Thus, abstract class cannot be tested, as its object can't be created.
2. The methods should be tested with reference to class or class hierarchy in the direction of complete testing of a class.
3. Another important thing is inheritance is some classes have been derived from the bare class, them dependent class testing is not sufficient.

Feature based testing :
The features of class are categorized into six groups.

- Create
- Destroy
- Moefiers
- Predicates
- Selectors
- Iterators

State based testing :

- State based testing is based on finite state customers that tests each feature with all its valid states.
- State based testing tests interactions by monitoring effects that feature have on state of object.
- The process of state based testing is:
    - Define the states.
    - Define the transistors between states.
    - Define test scenarios.
    - Define test values for each state.

Example for class testing: class to implement an integer stack class stack with following data item count, empty, full, pop max size.
So for testing this class following points must be considered.

1. Item count cannot have negative.
2. There is upper bound check on item count. I.e. item count ≤ max size.
3. Empty stack i.e. item count = 0
4. Full stack i.e. item count = max size.

OBJECT ORIENTED SOFTWARE ENGINEERING

UNIT-4

Modelling with UML

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.

UML was created by the Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997.

OMG is continuously making efforts to create a truly industry standard.

- UML stands for **Unified Modeling Language**.
- UML is different from the other common programming languages such as C++, Java, COBOL, etc.
- UML is a pictorial language used to make software blueprints.
- UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.
- Although UML is generally used to model software systems, it is not limited within this boundary. It is also used to model non-software systems as well. For example, the process flow in a manufacturing unit, etc.

UML is not a programming language but tools can be used to generate code in various languages using UML diagrams. UML has a direct relation with object oriented analysis and design. After some standardization, UML has become an OMG standard.

Goals of UML

A picture is worth a thousand words, this idiom absolutely fits describing UML. Object-oriented concepts were introduced much earlier than UML. At that point of time, there were no standard methodologies to organize and consolidate the object-oriented development. It was then that UML came into picture.

There are a number of goals for developing UML but the most important is to define some general purpose modeling language, which all modelers can use and it also needs to be made simple to understand and use.

UML diagrams are not only made for developers but also for business users, common people, and anybody interested to understand the system. The system can be a software or non-software system. Thus it must be clear that UML is not a development method rather it accompanies with processes to make it a successful system.

In conclusion, the goal of UML can be defined as a simple modeling mechanism to model all possible practical systems in today’s complex environment.

Basic building blocks of UML

As UML describes the real-time systems, it is very important to make a conceptual model and then proceed gradually. The conceptual model of UML can be mastered by learning the following three major elements −

- UML building blocks
- Rules to connect the building blocks
- Common mechanisms of UML

This chapter describes all the UML building blocks. The building blocks of UML can be defined as −

- Things
- Relationships
- Diagrams

Things

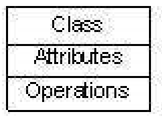
Things are the most important building blocks of UML. Things can be −

- Structural
- Behavioral
- Grouping
- Annotational

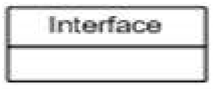
Structural Things

Structural things define the static part of the model. They represent the physical and conceptual elements. Following are the brief descriptions of the structural things.

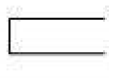
Class − Class represents a set of objects having similar responsibilities.



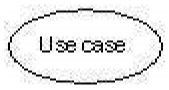
Interface − Interface defines a set of operations, which specify the responsibility of a class.



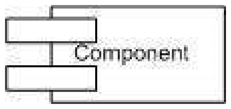
Collaboration −Collaboration defines an interaction between elements.



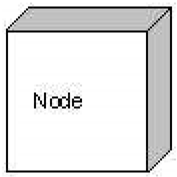
Use case −Use case represents a set of actions performed by a system for a specific goal.



Component −Component describes the physical part of a system.



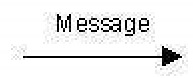
Node − A node can be defined as a physical element that exists at run time.



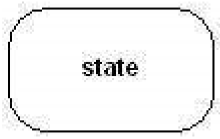
Behavioral Things

A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things −

Interaction − Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.



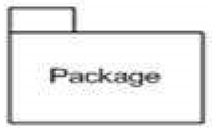
State machine − State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change



Grouping Things

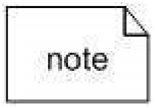
Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available −

Package − Package is the only one grouping thing available for gathering structural and behavioral things.



Annotational Things

Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. **Note** - It is the only one Annotational thing available. A note is used to render comments, constraints, etc. of an UML element.



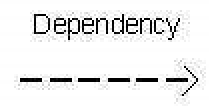
Relationship

Relationship is another most important building block of UML. It shows how the elements are associated with each other and this association describes the functionality of an application.

There are four kinds of relationships available.

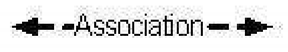
Dependency

Dependency is a relationship between two things in which change in one element also affects the other.



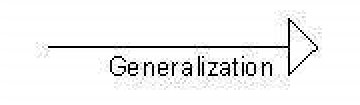
Association

Association is basically a set of links that connects the elements of a UML model. It also describes how many objects are taking part in that relationship.



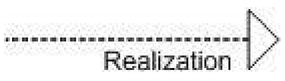
Generalization

Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes the inheritance relationship in the world of objects.



Realization

Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility, which is not implemented and the other one implements them. This relationship exists in case of interfaces.



UML Diagrams

UML diagrams are the ultimate output of the entire discussion. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system.

The visual effect of the UML diagram is the most important part of the entire process. All the other elements are used to make it complete.

UML includes the following nine diagrams, the details of which are described in the subsequent chapters.

- Class diagram
- Object diagram
- Use case diagram
- Sequence diagram
- Collaboration diagram
- Activity diagram
- Statechart diagram
- Deployment diagram
- Component diagram

A Conceptual Model of UML

To understand the conceptual model of UML, first we need to clarify what is a conceptual model? and why a conceptual model is required?

- A conceptual model can be defined as a model which is made of concepts and their relationships.
- A conceptual model is the first step before drawing a UML diagram. It helps to understand the entities in the real world and how they interact with each other.

As UML describes the real-time systems, it is very important to make a conceptual model and then proceed gradually. The conceptual model of UML can be mastered by learning the following three major elements −

- UML building blocks
- Rules to connect the building blocks
- Common mechanisms of UML

Object-Oriented Concepts

UML can be described as the successor of object-oriented (OO) analysis and design.

An object contains both data and methods that control the data. The data represents the state of the object. A class describes an object and they also form a hierarchy to model the real-world system. The hierarchy is represented as inheritance and the classes can also be associated in different ways as per the requirement.

Objects are the real-world entities that exist around us and the basic concepts such as abstraction, encapsulation, inheritance, and polymorphism all can be represented using UML.

UML is powerful enough to represent all the concepts that exist in object-oriented analysis and design. UML diagrams are representation of object-oriented concepts only. Thus, before learning UML, it becomes important to understand OO concept in detail.

Following are some fundamental concepts of the object-oriented world −

- **Objects** − Objects represent an entity and the basic building block.
- **Class** − Class is the blue print of an object.
- **Abstraction** − Abstraction represents the behavior of an real world entity.
- **Encapsulation** − Encapsulation is the mechanism of binding the data together and hiding them from the outside world.
- **Inheritance** − Inheritance is the mechanism of making new classes from existing ones.
- **Polymorphism** − It defines the mechanism to exists in different forms.

OO Analysis and Design

OO can be defined as an investigation and to be more specific, it is the investigation of objects. Design means collaboration of identified objects.

Thus, it is important to understand the OO analysis and design concepts. The most important purpose of OO analysis is to identify objects of a system to be designed. This analysis is also done for an existing system. Now an efficient analysis is only possible when we are able to start thinking in a way where objects can be identified. After identifying the objects, their relationships are identified and finally the design is produced.

The purpose of OO analysis and design can described as −

- Identifying the objects of a system.
- Identifying their relationships.
- Making a design, which can be converted to executables using OO languages.

There are three basic steps where the OO concepts are applied and implemented. The steps can be defined as

```
OO Analysis → OO Design → OO implementation using OO languages
```

The above three points can be described in detail as −

- During OO analysis, the most important purpose is to identify objects and describe them in a proper way. If these objects are identified efficiently, then the next job of design is easy. The objects should be identified with responsibilities. Responsibilities are the functions performed by the object. Each and every object has some type of responsibilities to be performed. When these responsibilities are collaborated, the purpose of the system is fulfilled.
- The second phase is OO design. During this phase, emphasis is placed on the requirements and their fulfilment. In this stage, the objects are collaborated according to their intended association. After the association is complete, the design is also complete.
- The third phase is OO implementation. In this phase, the design is implemented using OO languages such as Java, C++, etc.

Role of UML in OO Design

UML is a modeling language used to model software and non-software systems. Although UML is used for non-software systems, the emphasis is on modeling OO software applications. Most of the UML diagrams discussed so far are used to model different aspects such as static, dynamic, etc. Now whatever be the aspect, the artifacts are nothing but objects.

If we look into class diagram, object diagram, collaboration diagram, interaction diagrams all would basically be designed based on the objects.

Hence, the relation between OO design and UML is very important to understand. The OO design is transformed into UML diagrams according to the requirement. Before understanding the UML in detail, the OO concept should be learned properly. Once the OO analysis and design is done, the next step is very easy. The input from OO analysis and design is the input to UML diagrams.

Basic Structural Modelling

A. Classes

1. Why class

- Most important building block of any object-oriented system.
- A description of a set of objects that share the same attributes, operations, relationships, and semantics.

- To capture the vocabulary of the system being developed in both application and implementation domains.

1. Terms and concepts

- Class name: (noun with first letter in capital)
  Simple name: Wall
  Path name: java::awt::Rectangle
- Class attribute:
  What is an attribute?
  An attribute is a named property of a class that describes a range of values that instances of the property may hold.
  Name convention: noun with first letter in lower case
  Definition of an attribute:
  name: Type = defaultValue
  e.g. height: float = 0.0
- Class operations
  What is an operation?
  An operation is the implementation of a service that can be requested from any object of the class to affect behavior.
  Name convention: verb with first letter in lower case
  Definition of an operation with its signature
  name(parameters): return type
  e.g. getValue(o: Object): integer
- Class responsibility
  What is responsibility?
  A responsibility is a contract or an obligation of
  obligation of a class.
  Cohesion rule:
  A class must have at least one, but very few responsibility.
  Class attributes and operations realize the class' responsibility.
  In UML, class responsibility can be specified as free text in associated documentation.

1. Common modeling techniques

- Modeling the vocabulary of a system
  a. Identify those things that users or implementers use to describe the problem or solution.
  b. For each abstraction, identify a set of responsibilities.
  c. Provide the attributes and operations that are needed
- Modeling the distribution of responsibilities in a system
  a. Identify a set of classes that work together closely to carry out some behavior
  b. Identify a set of responsibilities for each of these classes
  c. Look at this set of class as a whole,

- Split classes that have too many responsibilities into small abstractions
- Collapse tiny classes that have trivial responsibilities into larger ones
- Reallocate responsibilities so that each abstraction reasonably stands on its own.
- Redistribute responsibilities for the collaboration to make each class not do too much or too little.

1. What is a well-structured class

- Provides a crisp abstraction of something drawn from the vocabulary of the problem domain or the solution domain.
- Embodies a small, well-defined set of responsibilities and carries them all out very well.
- Provides a clear separation of the abstraction's specification and its implementation.
- Is understandable and simple yet extensible and adaptable.

A. Relationships between classes

1. Terms and Concepts

- Dependency

  A dependency is a using relationship that states that a change in specification of one thing may affect another thing that uses it, but not necessarily the reverse.
  Common dependency in OO:
  Class A depends on class B if B is referenced in A as a type name of a class attribute or a parameter of an operation.
- Generalization

  A generalization is a relationship between a general thing (superclass or parent) and a more specific kind of that thing (subclass or child).
  Generalization is an "*is-a-kind-of*" relationship.
  Generalization means:
  - A child inherits its parent's attributes and operations. (inheritance)
  - A child can redefine an operation inherited form its parent. (polymorphism)
  - A child is substitutable for the parent, but not reverse. (subtype)

  Terms:
  - Root class: a class has no parent but has child
  - Leaf class: a class has no child
  - Single inheritance: a child has only one parent
  - Multiple inheritance: a child has two or more parent.
- Association

  An association is a structural relationship that specifies that objects of one class are connected to objects of another class.

Elements of an association

- Name:
  describes meaning of the association
- Role name:
  describes what is the role of each participating class in the association.
- Multiplicity
  describes, for an object of one participating class, how many (range) objects of the other participating class it can connect to.
- Aggregation
  describes a "whole/part" or "has-a" relationship, i.e. an object of the whole has objects of the part.
  Plain association: peer to peer
  Aggregation: whole to part

1. Common modeling techniques

- Modeling simple dependencies
  Create a dependency pointing from the class with the operation to the class used as a parameter in the operation.
- Modeling single inheritance

a. Given a set of classes, look for responsibilities, attributes, and operations that common to two or more classes.
b. Elevate these common responsibilities, attributes, and operations to a more general class.
c. Specify the more-specific classes inherited from the more-general class with generalization relationship.

- Modeling structural relationship

a. Identify association using data-data-driven view Objects of class A send data to objects of class B
b. Identify association using behavior-driven view Objects of class A need to interact with objects of class B
c. Set up roles and multiplicity
d. Identify whole/part or has-a relationship

C. Common Mechanisms

1. Terms and Concepts

- Notes
  A note is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements.
- Named compartment

- Stereotype
  A stereotype is an extension of the vocabulary of the UML, to allow creating new kinds of building blocks.
- Tagged Values
  A tagged value is en extension of the properties of a UML element to allow creating new information in the specification of that element.
  A tagged value is metadata.
- Constraints
  A constraint is an extension of the semantics of a UML element to allow adding or modifying rules.

1. Common Modeling Techniques

- Modeling Comments

1. Put a comment in a note and link the note to the corresponding
   element using dependency.
2. Make a note visible or invisible
3. Embed external document in a note.

- Modeling New Building Block with Standard Blocks and Stereotypes
- Modeling New Properties for a UML element
  Using stereotypes, and tagged values
- Modeling New Semantics for a UML element using constraints

D. UML Diagrams

1. Terms and Concepts

- Structural Diagrams
- **Class diagram** shows classes, interfaces, and their collaboration relationships
  For illustrating the static design view of a system.
- **Object diagram** shows a set of objects and their relationships.

For illustrating data structures, the static snapshots of class instances.

- **Component Diagram** shows a set of components and their relationships.
  For illustrating the static implementation view of a system
- **Deployment diagram** shows a set of nodes and their relationships.
  For illustrating the static deployment view of an architecture.
- **Behavioral Diagrams**
- **Use case diagram** shows a set of use cases and actors and their relationships.
  For illustrating the static use case view of a system.
- **Sequence diagram** shows a set of objects and messages sent and received by those objects, with emphasis on the time ordering of messages.
  For illustrating the dynamic view of a system
- **Collaboration diagram** shows a set of objects that send and receive messages, with emphasis on the structural organization of the collaboration objects.
  For illustrating the dynamic view of a system
- **Statechart diagram** shows a state machine of a class, consisting of states, transitions, and activities, with emphasis on the event-ordered behavior of an object.
  For illustrating the dynamic view of a system
- **Activity diagram** shows the flow from activity to activity within a system.
  For illustrating the dynamic view of a system

1. Common Modeling Techniques

- Modeling Different Views of a System
- Views
  Use case view:
  The use of use cases that describe the behavior of the system as seen by its end users, analysts, and testers.
  Design View:
  The classes, interfaces, and collaborations that form the vocabulary of the problem and its solution.
  Supporting the functional requirements of the system.
  Process View:
  The threads and processes that form the concurrency and synchronization of the system.
  Addressing the performance, scalability, and throughput of the system.
  Implementation View:
  The components and files that are used to assemble and release the physical system
  Addressing the configuration of the system.
  Deployment View:
  The nodes that form the hardware topology of the system on which the system executes
  Addressing the distribution, delivery, and installation of parts that make up the physical system.
- Modeling the views using UML diagrams
  Use case view:
  Use case diagrams, interaction diagrams, activity diagrams

Design view:
Class diagrams, interaction diagrams, statechart diagrams
Process view:
Class diagrams (with active classes), interaction diagrams
Implementation view:
Component diagrams
Deployment view
Deployment diagram

- Simple monolithic application

Use case view, design view

- Client/server systems

  Use case view, design view, component view, deployment view
- Modeling Different Levels of Abstraction
- Presenting diagrams with different levels of detail in a single model
- Creating models at different levels of abstraction

E. Class Diagrams

1. Terms and Concepts

- Contents of class diagrams
- Classes
- Interfaces
- Collaborations
- Dependency, Generalization, and association relationships
- Notes
- Constraints
- Packages
- Subsystems

1. Common Modeling Techniques

- Modeling Simple Collaborations

1. Identify the mechanism to be modeled
   Mechanism represents some function or behavior of the part of the system from interaction of classes, interfaces, and others.
2. For each mechanism, identify the participating classes, interfaces, and others.
3. Use scenarios to walk through these things to discover missing and wrong.

- Modeling a Logical Database Schema

1. Identify those classes whose state must transcend the lifetime of their applications.
2. Create a class diagram that contains these classes and mark them as persistent (a standard tagged value)
3. Specify the details of their attributes, associations, and cardinalities.
4. Use intermediate abstractions to solve cyclic, one-to-one, and n-ary associations.
5. Expanding operations for data access and data integrity. (No business rules in this layer)

- Forward and Reverse Engineering
- Forward engineering

  The process of transforming a model into code through a mapping to an implementation language.

  Result in a loss of information (structural and behavioral features)

  Forward engineering a class diagram:

  1. Identify the rules for mapping to the implementation language
  2. Constrain the use of certain UML features because of language limitation
  3. Use tagged values to specify the target language
  4. Use a tool to forward engineer the model.

  - Reverse engineering

    The process of transforming code into a model through a mapping from the implementation language.

    Result in a flood of low-level information.

Basic Behavioral Modelling

A. Interactions

1. Terms and Concepts
- Interaction
  A behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose.
- Message
  A specification of a communication between objects that conveys information with the expectation that activity will ensue.
- Objects and Roles
- An object participating an interaction can be a concrete instance or a prototype instance of a class
- A prototype instance of an abstract class represents a prototype of any subclass
- Links
  A link is an instance of an association denoting a semantic connection between objects.
  Stereotypes for visibility from sender to receiver
  Association, self, global, local, parameter
- Messages
  A message can trigger an action.
  Kinds of actions:
- **Call** invokes an operation on the receiver
- **Return** a value to the sender
- **Create** a new object
- **Destroy** a existing object
- Sequencing
- Procedural sequencing: nested ordered messages
- Flat sequencing: linearly ordered messages

1. Common Modeling Techniques
- Modeling a Flow of Control

a. Set the context got the interaction
b. Identify participating objects and set their roles
c. Identify links between objects
d. Specify the messages that pass from object to object in time order with the associated arguments and return values.

- Flow control by Time

- Flow control by Organization

B. Use Cases

1. Terms and Concepts
- Use case
  A description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor.
- Names: simple, path
- Use cases and actors
  An actor represents a coherent set of roles that users of use cases play when interacting with these use cases.
  Actor/actor relationship: generalization

Actor/use case relationship: association

- Use case and flow of events
- Use case: WHAT
- Event flow: HOW
- Event flows: main, exceptional (alternative)
- Use cases and scenarios
- A scenario is an instance of a use case.
- A scenario consists of a set of interactions, each an instance of a (sub) event flow of the use case.
- Use cases and Collaborations

  A use case can be implemented by a collaboration
- Organizing use cases
- Packages
- Generalization relationship
- Include relationship (for common behavior)

  The base use case explicitly incorporates the behavior of another use case at a location specified in the base

  The base use case delegates part of work to the included use case
- Extend relationship (for system options/exceptions)

  The base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case

  The behavior of the base is extended by the behavior of the extending use case.

  The base class can only be extended at its extension points

  The extension use case push behavior to the base use case

2. Common Modeling Techniques

- Modeling the behavior of an Element

a. Identify the actors that interact with the element.
b. Organize actors by identifying general and more specialized roles.
c. For each actor, consider the primary ways in which the actor interact with the element
d. Consider exceptional ways in which each actor interacts with the element
e. Organize these behaviors (ways) as use cases

C. Use Case Diagrams

1. Terms and Concepts

- Contents
- Use cases
- Actors
- Dependency, generalization, association relationships
- Packages

1. Common Modeling Techniques

- Modeling the Context of a System

a. Identify actors by grouping user groups
    - o Those require help from the system to perform their tasks
    - o Those are needed to execute the system functions

a. Organize actors
b. Provide stereotype for each actor
c. Specify the paths of communication from each actor to the system use cases

- Modeling the Requirements of a System

a. Identify the actors to establish the system context
b. Consider each what behaviors each actor expect or require the system to provide

c. Name these common behaviors as use cases
d. Organize the use cases using relationships

- Forward Engineering

  Form test cases from a use case diagram

  For each use case, create a test case

  a. Identify the event flows (main, exceptional) of the use case
  b. Generate a test script for each flow
  c. Generate simulated actor(s)

D. Interaction Diagrams

1. Terms and Concepts

- Contents
- Objects
- Links
- Messages

- Sequence Diagram

  Emphasize the time ordering of messages

  Special features
- Show object lifetime
- Show the focus of control:

  A period of time during which an object is performing an action directly or indirectly.
- Collaboration Diagrams

  Emphasize the organization of the objects that participate in an interaction

  Distinct features:
- Show linking path
- Show procedural sequence number

  Sequence diagram and collaboration diagram is semantically equivalent

1. Common Modeling Techniques

- Modeling Flows of Control by Time Ordering

a. Set the context for the interaction
b. Identify participating objects/roles
c. Set the lifeline for each object
d. Start the message that initiates this interaction from the top
e. Adorn the lifeline of each object with its focus of control
f. Adorn timing marks with time/space constraints

- Modeling Flow of Control by Organization

a. Same as sequence diagrams in first three steps
b. For an object whose state will be changed during this interaction, create a duplicated object with new state and connected it with a message stereotyped as <<become>> or <<copy>>
c. Lay out association links
d. Lay out other links
e. Attach each message to the appropriate links with (nested) sequence numbers.
a. Attach pre/post-conditions

E. State Machines and Statecharts

1. Terms and Concept

- State machine

A behavior that specifies the sequence of states an object goes through during its lifetime in response to evens, together with its responses to those events.

- State

  A condition or situation during the life of an object during which it satisfies some conditions, perform some activities, or waits for some events.

  Basic Parts:
    - Name
    - Entry/exit actions
- Event

  An occurrence of a stimulus that cab trigger a state transition.
- Transition

  A relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified condition occurs.

  Parts:
    - Source state
    - Target state
    - Event trigger
    - Guard condition
    - Action

1. Common Modeling Techniques

- Modeling Reactive Objects

  Example: Parsing a simple context-free language

- Forward engineering

  Code generation for a class from its (detailed) statechart

Architectural Modelling

A. Components

1. Terms and Concepts

- Component
  A physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.
- Components and classes
  - Classes represent logical abstractions
    Components represent physical things
  - Components represent the physical packaging of logical components at implementation level
  - Components can have operations that are reachable only through their interfaces

- Components and interfaces
  - Interface
    A collection of operations (not realization of the operations) that used to specify a service of a class or a component
  - Exporting interface of a component
    The component realizes the interface to provide the services.
    A component can realize more than one interface
  - Importing interface of a component
    The component uses the services provided by the interface to do the realization
    A component can import more than one interface

- Kinds of components
  - Deployment components (dll, exe)
    Form an executable system
  - Work product components (source file, data file)
    Residue of the development process
  - Execution component
    Created as a consequence of an executing system
- Organizing components
  Package and relationships (dependency, generalization, association)
- Standard UML stereotypes for components
  - Executable
  - Library
  - Table
  - File
  - Document

1. Common Modeling Techniques

- Modeling Executables and Libraries

a. Identify the partitioning of the physical system
b. Model any executables and libraries as components
c. Model the significant interfaces
d. Model the relationships among the executables, libraries, and interfaces

- Modeling Tables, Files, and Documents

a. Identify the ancillary components of the system

b. Model them as components
c. Model their relationships

- Modeling and API

a. Identify the programmatic seams in the system
b. Model each seam as an interface
c. Collection the attributes and operations that form the interface
d. Expose only the properties of the interface that are important to visualize in the given context

- Modeling Source Code

B. Deployment
1.Terms and Concepts

- Node
  A physical element that exists at run time and represents a computational resource
- Nodes and components
  Node Component
  Execute components Participate in the execution
  Represent physical deployment Represent physical packaging
  Of components of logical elements
  - Distribution unit
    A set of objects or components that are allocated to a node as a group

- Connections
  Association relationships among nodes

- Modeling the Distribution of Components

a. List each component deployed on a node in an additional compartment
b. Using dependency relationships to connect each node with the components it displays
c. Using association relationships to connect nodes

C. Component Diagrams

- Contents
  Components, interfaces, relationships (dependency, generalization, association, and realization)
- Common uses
  - To model source code
  - To model executable release
  - To model physical database (mapping classes to tables)
  - To model adaptive systems (migration, duplication)

D. Deployment Diagrams

- Contents
  - Nodes
  - Dependency and association relationships

- Modeling a Client/Server System

a. Identify the nodes the represent the client and server processors
b. Highlight special devices other than computers
c. Provide visual cues for the processors and devices
d. Model the topology of the nodes in a deployment program

- Modeling a Fully Distributed System

a. Identify the processors and devices
b. Model communication devices to the desired detail level for performance evaluation
c. Model the underlying network as a node
d. Model the devices and processors using deployment diagrams
e. Create use case diagrams and interaction diagrams to specify the system interactions