UNIT -1 : OBJECT-ORIENTED SOFTWARE ENGINEERING - OOSE

Object-Oriented Software Engineering (OOSE) is a software design technique that is used in software design in object-oriented programming.

OOSE is developed by **Ivar Jacobson** in 1992. OOSE is the first object-oriented design methodology that employs use cases in software design. OOSE is one of the precursors of the Unified Modeling Language (UML), such as Booch and OMT.

It includes a requirements, an analysis, a design, an implementation and a testing model.

Interaction diagrams are similar to UML's sequence diagrams. State transition diagrams are like UML statechart diagrams.

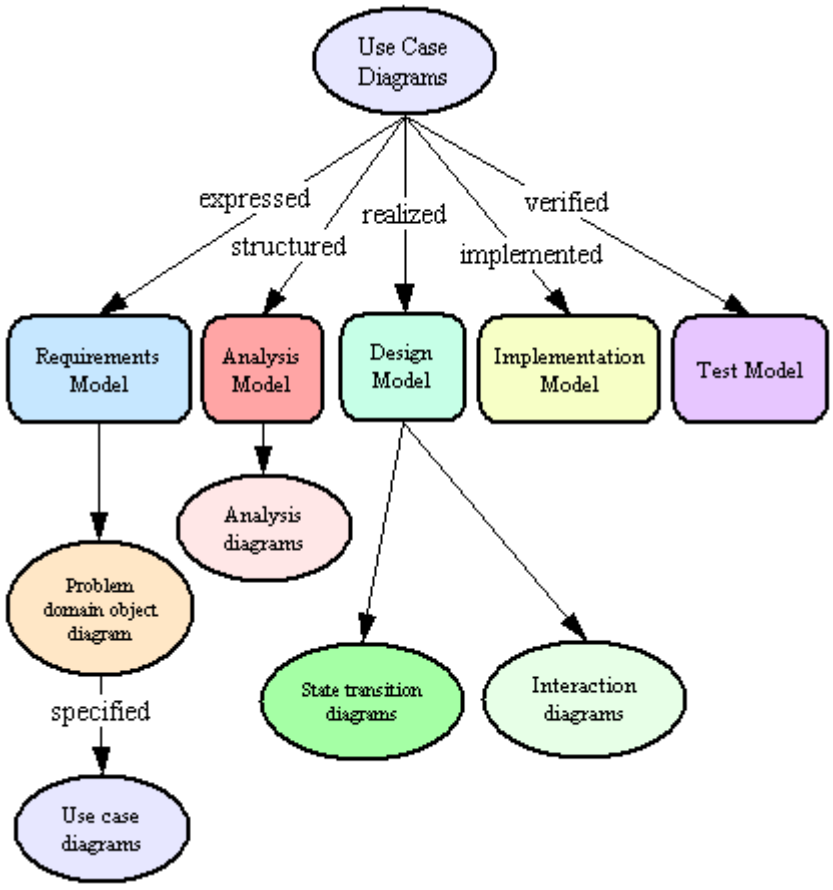


Figure 1. Object-Oriented Software Engineering.

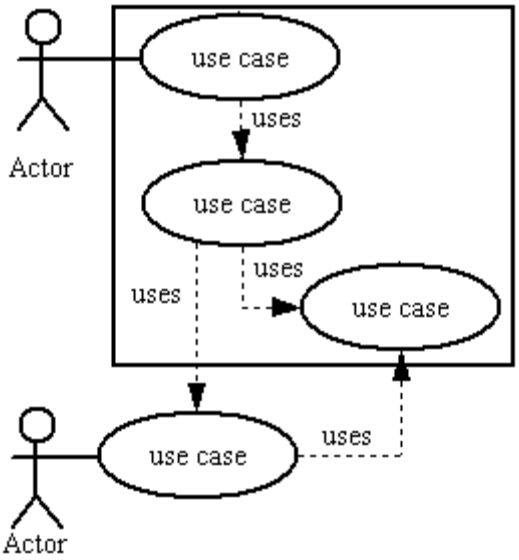


Figure 2. Jacobson's Use Case diagram.

Below, we'll take a look at the biggest challenges for software developers in 2021 and what they can do to overcome them.

1. Keeping Pace with Innovation. ...
2. Cultural Change. ...
3. Customer Experience. ...
4. Data Privacy. ...
5. Cybersecurity. ...
6. AI and Automation. ...
7. Data Literacy. ...
8. Cross-Platform Functionality.
9. Budgeting
10. .Talent

SOFTWARE DEVELOPMENT PROCESS STEPS

The software development process consists of four major steps. Each of these steps is detailed below.

- Step 1: Planning
- Step 2: Implementing
- Step 3: Testing
- Step 4: Deployment and Maintenence

Step #1: Planning

An important task in creating a software program is Requirements Analysis. Customers typically have an abstract idea of what they want as an end result, but not what software should do. Skilled and experienced software engineers recognize incomplete, ambiguous, or even contradictory requirements at this point. Frequently demonstrating live code may help reduce the risk that the requirements are incorrect. Once the general requirements are gathered from the client, an analysis of the scope of the development should be determined and clearly stated. This is often called a Statement of Objectives (SOO). [1]

Step #2: Implementation

Implementation is the part of the process where software engineers actually program the code for the project.

Step #3: Testing

Software testing is an integral and important phase of the software development process. This part of the process ensures that defects are recognized as soon as possible. It can also provide an objective, independent view of the software to allow users to appreciate and understand the risks of software deployment. Software testing can be stated as the process of validating and verifying that a software program/application/product: [1,2]

- meets the requirements that guided its design and development;
- works as expected; and
- can be implemented with the same characteristics.

Step #4: Deployment and Maintenance

Deployment starts after the code is appropriately tested, approved for release, and sold or otherwise distributed into a production environment. This may involve installation, customization, testing, and possibly an extended period of evaluation. Software training and support are important, as the software is only effective if it is used correctly. Maintaining and enhancing software to cope with newly discovered faults or requirements can take substantial time and effort, as missed requirements may force a redesign of the software. [1]

Software Development Plan (SDP)

The Software Development Plan (SDP) describes a developer's plans for conducting a software development effort. The SDP provides the acquirer insight and a tool for monitoring the processes to be followed for software development. It also details methods to be used and the approach to be followed for each activity, organization, and resource. The software development process should be detailed in the SDP.

SOFTWARE PROCESS MODEL

Software Processes is a coherent set of activities for specifying, designing, implementing and testing software systems. A software process model is an abstract representation of a process that presents a description of a process from some particular perspective. There are many different software processes but all involve:

- Specification – defining what the system should do;
- Design and implementation – defining the organization of the system and implementing the system;
- Validation – checking that it does what the customer wants;
- Evolution – changing the system in response to changing customer needs.

Types of Software Process Model

Software processes, methodologies and frameworks range from specific prescriptive steps that can be used directly by an organization in day-to-day work, to flexible frameworks that an organization uses to generate a custom set of steps tailored to the needs of a specific project or group. In some cases a “sponsor” or “maintenance” organization distributes an official set of documents that describe the process.

Software Process and Software Development Lifecycle Model

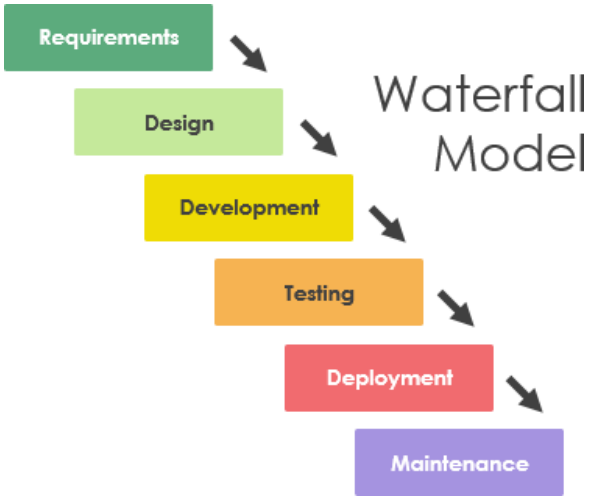
One of the basic notions of the software development process is SDLC models which stands for Software Development Life Cycle models. There are many development life cycle models that have been developed in order to achieve different required objectives. The models specify

the various stages of the process and the order in which they are carried out. The most used, popular and important SDLC models are given below:

- Waterfall model
- V model
- Incremental model
- RAD model
- Agile model
- Iterative model
- Spiral model
- Prototype model

Waterfall Model

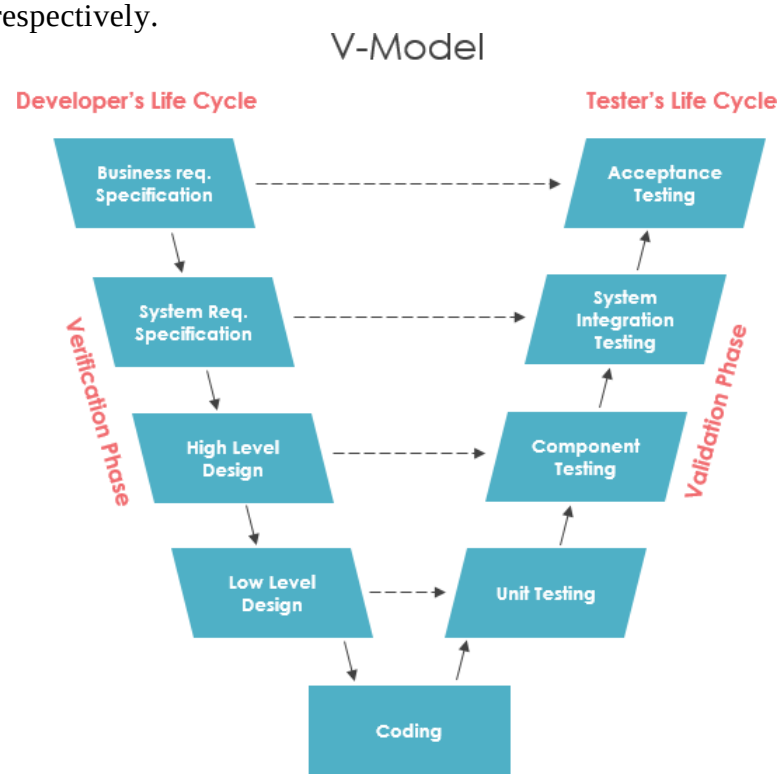
The waterfall model is a breakdown of project activities into linear sequential phases, where each phase depends on the deliverables of the previous one and corresponds to a specialisation of tasks. The approach is typical for certain areas of engineering design.



V Model

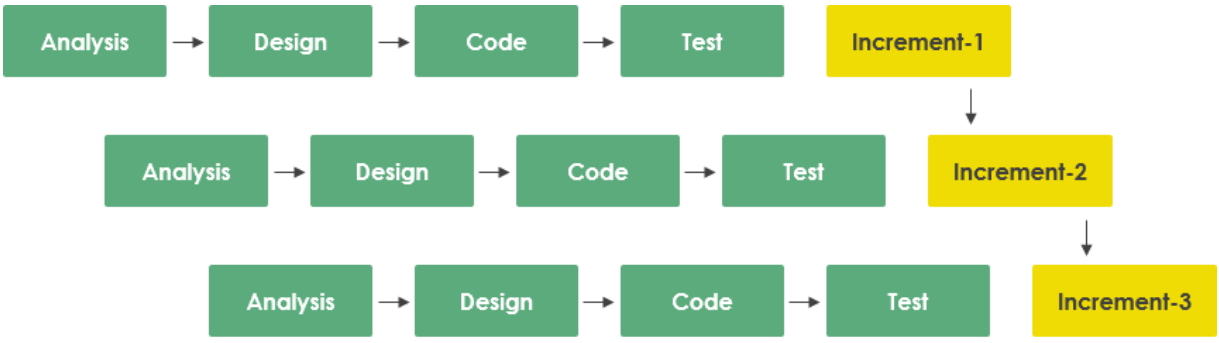
The V-model represents a development process that may be considered an extension of the waterfall model and is an example of the more general V-model. Instead of moving down in a linear way, the process steps are bent upwards after the coding phase, to form the typical V

shape. The V-Model demonstrates the relationships between each phase of the development life cycle and its associated phase of testing. The horizontal and vertical axes represent time or project completeness (left-to-right) and level of abstraction (coarsest-grain abstraction uppermost), respectively.



INCREMENTAL MODEL

The incremental build model is a method of software development where the model is designed, implemented and tested incrementally (a little more is added each time) until the product is finished. It involves both development and maintenance. The product is defined as finished when it satisfies all of its requirements. Each iteration passes through the requirements, design, coding and testing phases. And each subsequent release of the system



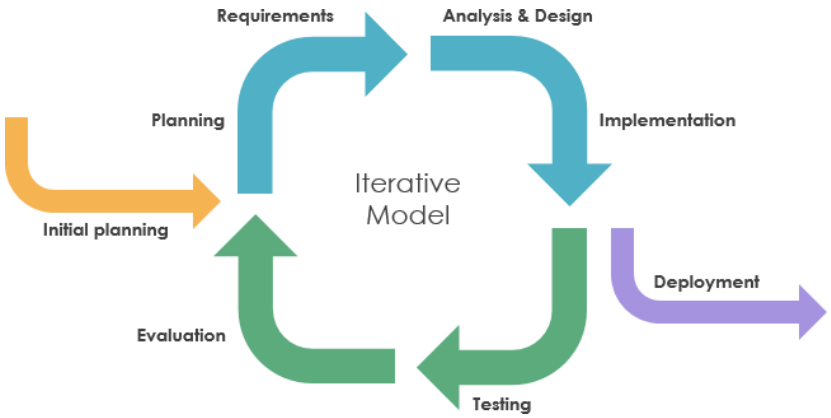
Incremental Model

adds function to the previous release until all designed functionally has been implemented. This model combines the elements of the waterfall model with the iterative philosophy of prototyping.

ITERATIVE MODEL

An iterative life cycle model does not attempt to start with a full specification of requirements by first focusing on an initial, simplified set user features, which then progressively gains more complexity and a broader set of features until the targeted system is complete. When adopting the iterative approach, the philosophy of incremental development will also often be used liberally and interchangeably.

In other words, the iterative approach begins by specifying and implementing just part of the software, which can then be reviewed and prioritized in order to identify further requirements. This iterative process is then repeated by delivering a new version of the software for each iteration. In a light-weight iterative project the code may represent the major source of documentation of the system; however, in a critical iterative project a formal software specification may also be required.

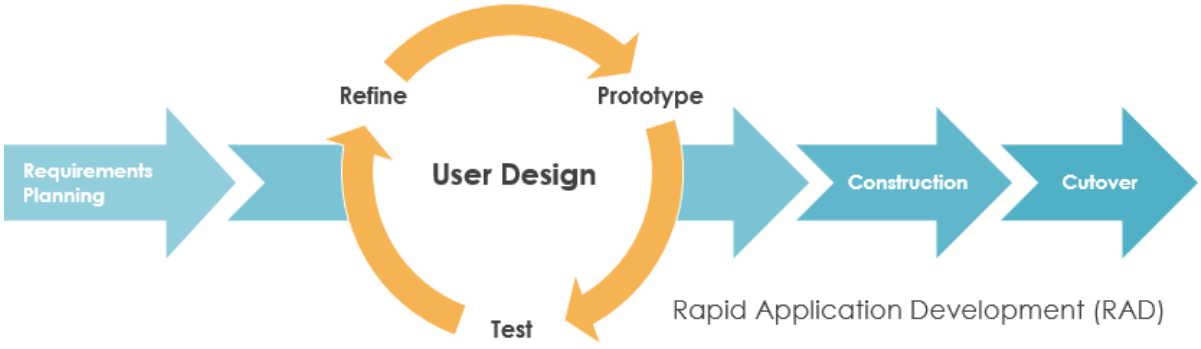


RAD MODEL

Rapid application development was a response to plan-driven waterfall processes, developed in the 1970s and 1980s, such as the Structured Systems Analysis and Design Method (SSADM). Rapid application development (RAD) is often referred as the adaptive software development. RAD is an incremental prototyping approach to software development that end users can produce better feedback when examining a live system, as opposed to working

strictly with documentation. It puts less emphasis on planning and more emphasis on an adaptive process.

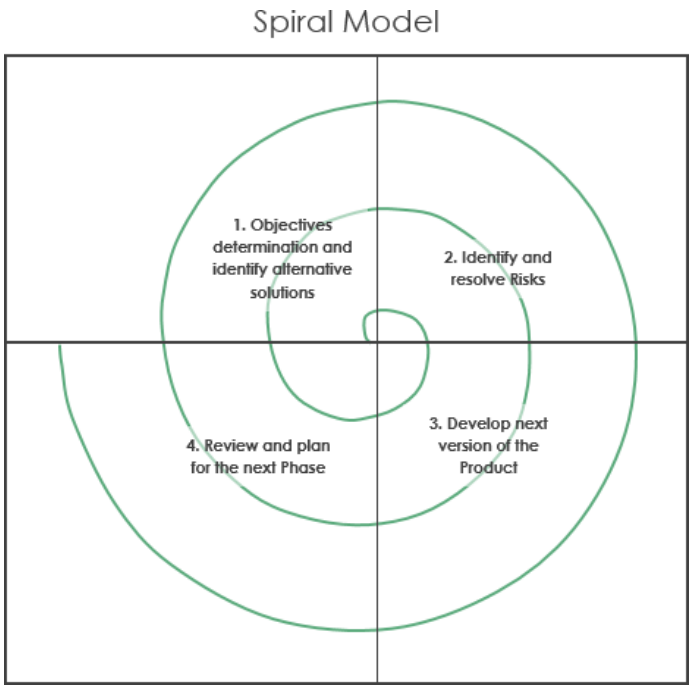
RAD may resulted in a lower level of rejection when the application is placed into production, but this success most often comes at the expense of a dramatic overruns in project costs and schedule. RAD approach is especially well suited for developing software that is driven by user interface requirements. Thus, some GUI builders are often called rapid application development tools.



SPIRAL MODEL

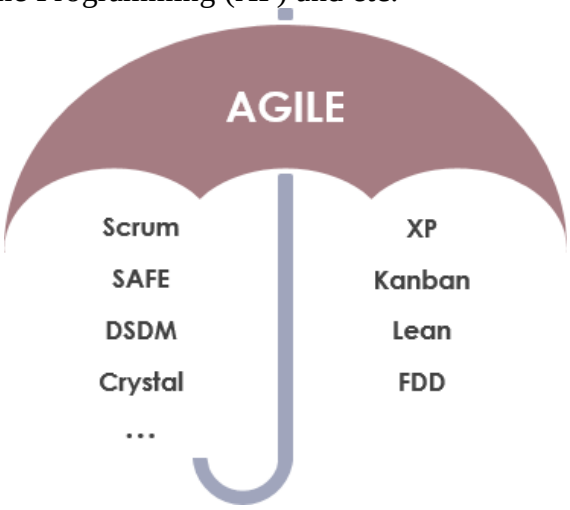
The spiral model, first described by Barry Boehm in 1986, is a risk-driven software development process model which was introduced for dealing with the shortcomings in the traditional waterfall model. A spiral model looks like a spiral with many loops. The exact number of loops of the spiral is unknown and can vary from project to project. This model supports risk handling, and the project is delivered in loops. Each loop of the spiral is called a Phase of the software development process.

The initial phase of the spiral model in the early stages of Waterfall Life Cycle that is needed to develop a software product. The exact number of phases needed to develop the product can be varied by the project manager depending upon the project risks. As the project manager dynamically determines the number of phases, so the project manager has an important role to develop a product using a spiral model.



AGILE MODEL

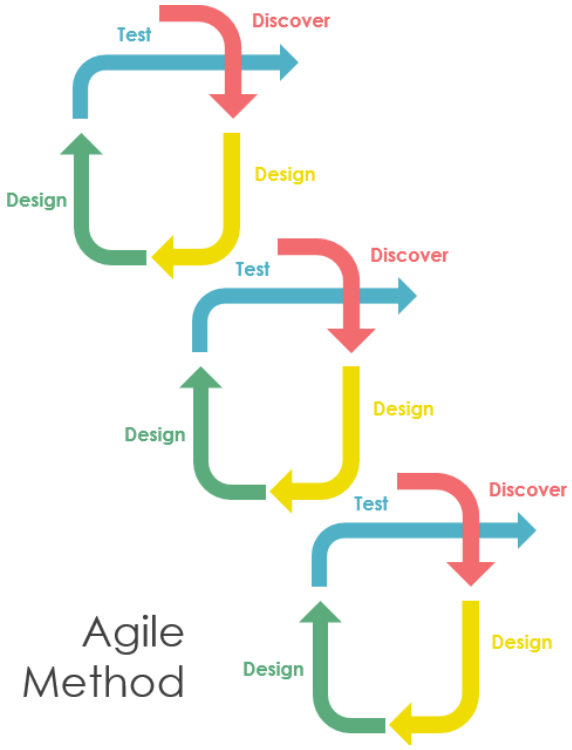
Agile is an umbrella term for a set of methods and practices based on the values and principles expressed in the Agile Manifesto that is a way of thinking that enables teams and businesses to innovate, quickly respond to changing demand, while mitigating risk. Organizations can be agile using many of the available frameworks available such as Scrum, Kanban, Lean, Extreme Programming (XP) and etc.



The Agile movement proposes alternatives to traditional project management. Agile approaches are typically used in software development to help businesses respond to

unpredictability which refer to a group of software development methodologies based on iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams.

The primary goal of being Agile is empowered the development team the ability to create and respond to change in order to succeed in an uncertain and turbulent environment. Agile software development approach is typically operated in rapid and small cycles. This results in more frequent incremental releases with each release building on previous functionality. Thorough testing is done to ensure that software quality is maintained.



UNIT -2: 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 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
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.

THE BOOCH METHOD - OOD

In software engineering the Booch method, that is published in 1991 by **Grady Booch**, is a widely used method in object-oriented analysis and design.

The Booch method has been superseded by UML, which features elements from the Booch method with OMT and OOSE.

The Booch method helps to design systems using the object paradigm. It covers the analysis- and design phases of an object-oriented system. The method defines different models to describe a system and it supports the iterative and incremental development of systems.

The Booch method includes **six types of diagrams** such as class diagrams, object diagrams, state transition diagrams, module diagrams, process diagrams and interaction diagrams.

The Booch method notation

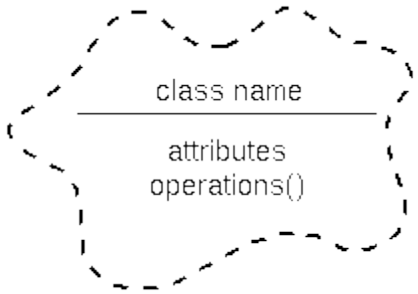


Figure 1. A class diagram notation.

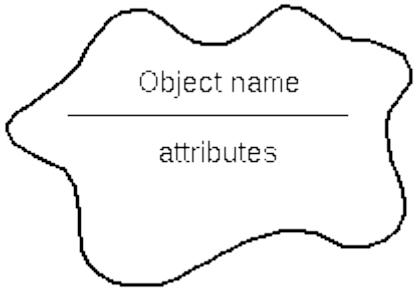


Figure 2. A object diagram notation.

The dynamic nature of an application can be illustrated by state transition and interaction diagrams.

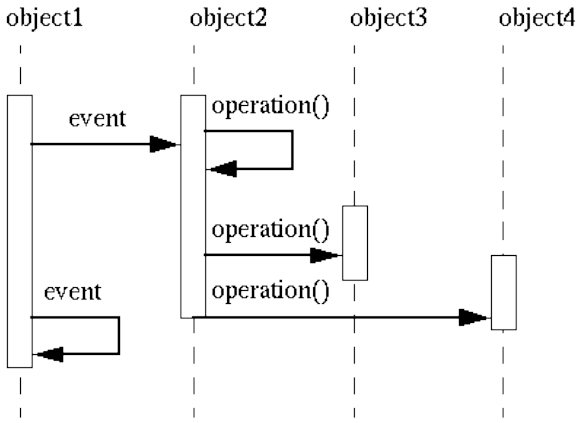
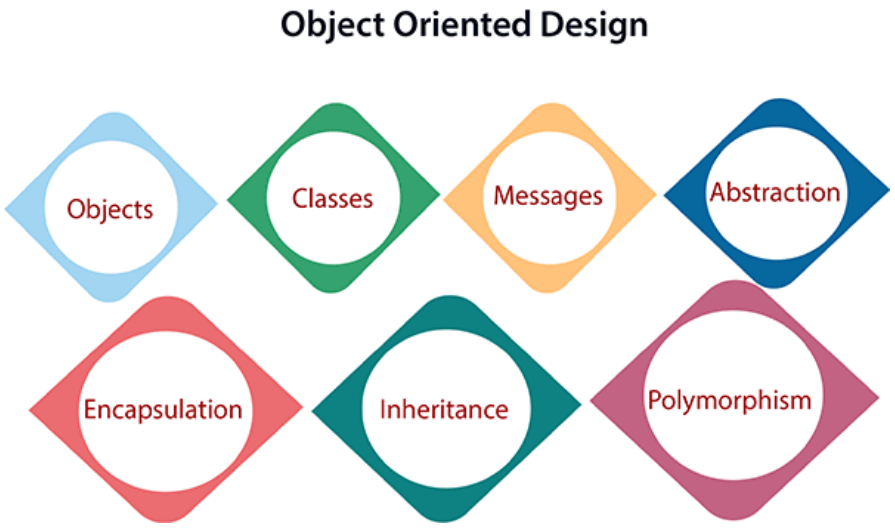


Figure 3. An interaction diagram.

There are several Booch diagrams that are very similar to diagrams in UML. These Booch diagrams are state transition and interaction diagrams. The State transition diagram corresponds to UML's statechart diagram and the interaction diagram corresponds to UML's sequence diagram.

OBJECT-ORIENTED DESIGN

In the object-oriented design method, the system is viewed as a collection of objects (i.e., entities). The state is distributed among the objects, and each object handles its state data. For example, in a Library Automation Software, each library representative may be a separate object with its data and functions to operate on these data. The tasks defined for one purpose cannot refer or change data of other objects. Objects have their internal data which represent their state. Similar objects create a class. In other words, each object is a member of some class. Classes may inherit features from the superclass.



The different terms related to object design are:

1. **Objects:** All entities involved in the solution design are known as objects. For example, person, banks, company, and users are considered as objects. Every entity has some attributes associated with it and has some methods to perform on the attributes.
2. **Classes:** A class is a generalized description of an object. An object is an instance of a class. A class defines all the attributes, which an object can have and methods, which represents the functionality of the object.
3. **Messages:** Objects communicate by message passing. Messages consist of the integrity of the target object, the name of the requested operation, and any other action needed to perform the function. Messages are often implemented as procedure or function calls.
4. **Abstraction** In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials.
5. **Encapsulation:** Encapsulation is also called an information hiding concept. The data and operations are linked to a single unit. Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.
6. **Inheritance:** OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate superclasses.This property of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.
7. **Polymorphism:** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name. This is known as polymorphism, which allows a single interface is performing functions for different types. Depending upon how the service is invoked, the respective portion of the code gets executed.

HIERARCHICAL OBJECT-ORIENTED DESIGN (HOOD)

developed from this by European Space Agency (ESA)

• The hierarchy described in HOOD takes two forms:

1. Uses: dependence of one object on another's services

2. Functional decomposition: object split into child objects, to give functionality of the parent object

10.1 HOOD theory

• Combination of data structures and functionality

– Object-oriented (OO) design was first used with OO

programming (e.g., Smalltalk)

– Based on concept of objects, classes and inheritance:

OO = Objects + Classes + Inheritance

Abstraction, Polymorphism, Encapsulation, Inheritance

1. Objects — abstraction

• operation is procedure provided by an object that

can access the internal data

• object is an information hiding module that contains

both data and operations on the data

• object is an instance of a Class of objects

2. Classes — modularity

• A class is an object template, like abstract data type

• Consists of encapsulated data types and operations

3. Inheritance — hierarchy

• A method of sharing and reusing code between classes:

– child class can adapt contents (data structure) and

behaviour (operations) of parent class by adding

operations and variables, or by redefining them

– new class is based on definition of existing class;

no need to copy actual code manually

– e.g., for designing two similar modules

• parent class is the super-class; child is the sub-class

• forms an “is a kind of” relationship: if class C inherits

from class B, then C “is a kind of” B

Reduce, recycle, re-use your computer code!

Objects can:

• have a common implementation so that they share code,

making implementation more efficient

• have different implementations of generic operations to

provide a consistent service — polymorphism

• share partial implementations (when selected elements

of the object’s behaviour are to be shared)

• be identified in a request, so when a client issues a

request it is clear which object should service it

• support parallel operations and time-varying behaviour

of a system

• Hierarchical Object-Oriented Design (HOOD) was

developed from this by European Space Agency (ESA)

• The hierarchy described in HOOD takes two forms:

1. Uses: dependence of one object on another's services

2. Functional decomposition: object split into child objects, to give functionality of the parent object

10.1 HOOD theory

• Combination of data structures and functionality

– Object-oriented (OO) design was first used with OO

programming (e.g., Smalltalk)

– Based on concept of objects, classes and inheritance:

OO = Objects + Classes + Inheritance

Abstraction, Polymorphism, Encapsulation, Inheritance

1. Objects — abstraction

• operation is procedure provided by an object that

can access the internal data

• object is an information hiding module that contains

both data and operations on the data

• object is an instance of a Class of objects

2. Classes — modularity

• A class is an object template, like abstract data type

• Consists of encapsulated data types and operations

3. Inheritance — hierarchy

• A method of sharing and reusing code between classes:

– child class can adapt contents (data structure) and

behaviour (operations) of parent class by adding

operations and variables, or by redefining them

– new class is based on definition of existing class;

no need to copy actual code manually

– e.g., for designing two similar modules

• parent class is the super-class; child is the sub-class

• forms an “is a kind of” relationship: if class C inherits

from class B, then C “is a kind of” B

Reduce, recycle, re-use your computer code!

Objects can:

• have a common implementation so that they share code,

making implementation more efficient

• have different implementations of generic operations to

provide a consistent service — polymorphism

• share partial implementations (when selected elements

of the object’s behaviour are to be shared)

• be identified in a request, so when a client issues a

request it is clear which object should service it

• support parallel operations and time-varying behaviour

of a system

USE CASES

In software and systems engineering, a use case is a list of actions or event steps, typically defining the interactions between a role (known in the Unified Modeling Language as an *actor*) and a system, to achieve a goal. The actor can be a human, an external system, or time. In systems engineering, use cases are used at a higher level than within software engineering, often representing missions or stakeholder goals. Another way to look at it is a use case describes a way in which a real-world actor interacts with the system. In a system use

case you include high-level implementation decisions. System use cases can be written in both an informal manner and a formal manner.

What is the importance of Use Cases?

Use cases have been used extensively over the past few decades. The advantages of Use cases includes:

- The list of goal names provides the shortest summary of what the system will offer
- It gives an overview of the roles of each and every component in the system. It will help us in defining the role of users, administrators etc.
- It helps us in extensively defining the user's need and exploring it as to how it will work.
- It provides solutions and answers to many questions that might pop up if we start a project unplanned.

How to plan use case?

Following example will illustrate on how to plan use cases:

Use Case: What is the main objective of this use case. For eg. Adding a software component, adding certain functionality etc.

Primary Actor: Who will have the access to this use case. In the above examples, administrators will have the access.

Scope: Scope of the use case

Level: At what level the implementation of the use case be.

Flow: What will be the flow of the functionality that needs to be there. More precisely, the work flow of the use case.

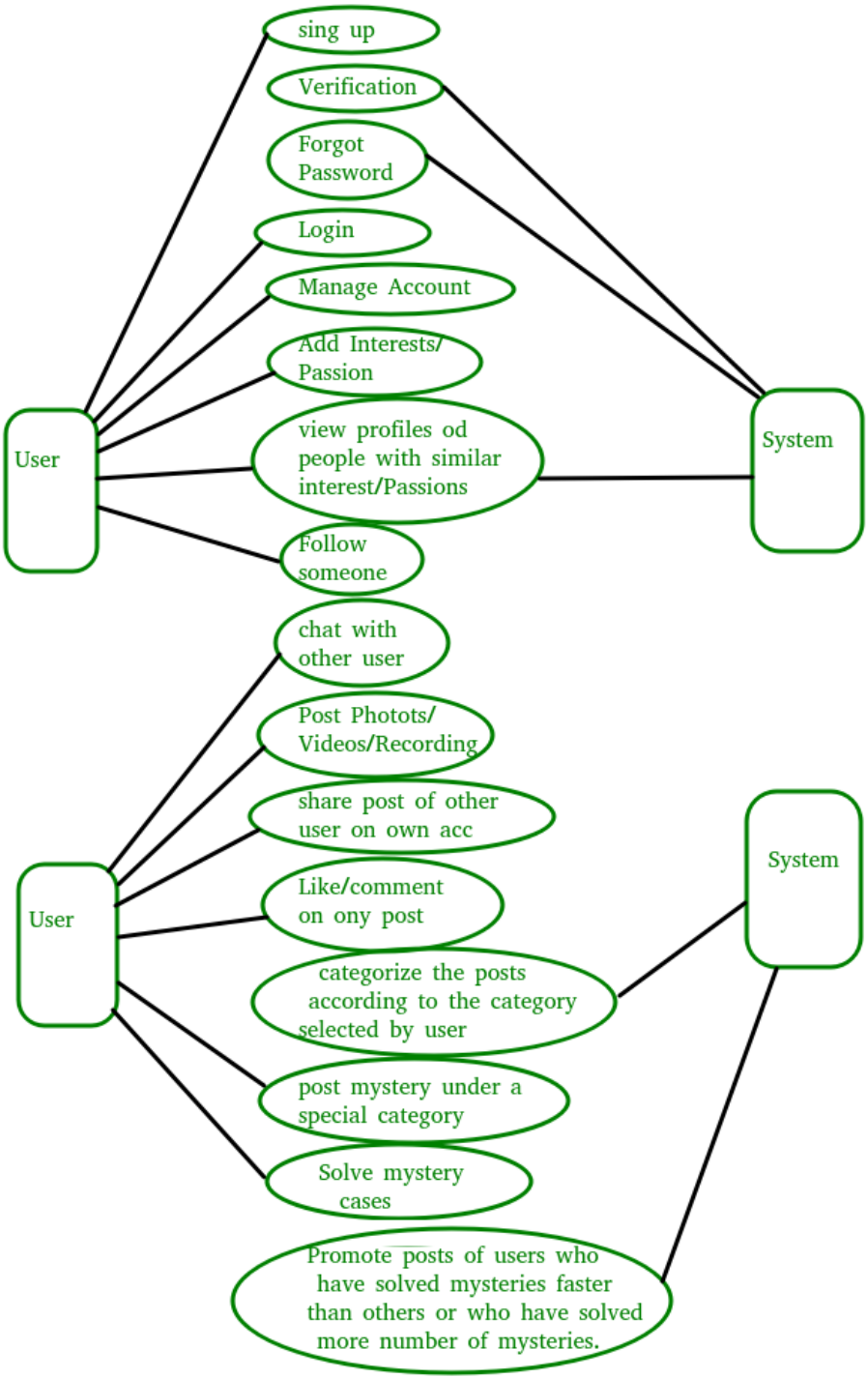
Some other things that can be included in the use cases are:

- **Preconditions**
- **Postconditions**
- **Brief course of action**
- **Time Period**

Use Case Diagram

Above is a sample use case diagram which I have prepared for reference purpose for a sample project (much like Facebook). It would help us to understand the role of various actors in our project. Various actors in the below use case diagram are: **User and System.**

The main use cases are in the system and the diagram illustrates on how the actors interact with the use cases.For eg. During Sign Up, only users need to interact with the use case and not the system whereas when it comes to categorizing posts, only system would be required.



SOFTWARE REQUIREMENTS SPECIFICATION (SRS) DOCUMENT

A software requirements specification (SRS) is a document that describes what the software will do and how it will be expected to perform. It also describes the functionality the product needs to fulfill all stakeholders (business, users) needs.

Why Use an SRS Document?

An SRS gives you a complete picture of your entire project. It provides a single source of truth that every team involved in development will follow. It is your plan of action and keeps all your teams — from development to maintenance — on the same page (no pun intended).

How to Write an SRS Document

Writing an SRS document is important. But it isn't always easy to do.

Here are five steps you can follow to write an effective SRS document.

1. Define the Purpose With an Outline (Or Use an SRS Template)

Your first step is to create an outline for your software requirements specification. This may be something you create yourself. Or you may use an existing SRS template.

If you're creating this yourself, here's what your outline might look like:

1. Introduction

- 1.1 Purpose
- 1.2 Intended Audience
- 1.3 Intended Use
- 1.4 Scope
- 1.5 Definitions and Acronyms

2. Overall Description

- 2.1 User Needs
- 2.2 Assumptions and Dependencies

3. System Features and Requirements

- 3.1 Functional Requirements
- 3.2 External Interface Requirements
- 3.3 System Features
- 3.4 Nonfunctional Requirements

CLASSES AND RELATIONSHIP

This is a basic outline and yours may contain more (or fewer) items. Now that you have an outline, lets fill in the blanks.

hat is a Class?

A Class is a description of a group of objects with common properties (attributes), behavior (operations), relationships, and semantics

A class is an abstraction. An object is an instance of a class.

Example of a Class

- Class: Course
- Properties: Name, Location, Days Offered, Credit Hours, Professor
- Behavior: Add Student, Delete Student, Get Course Roster, Determine If Full

A class is represented by a compartmentalized rectangle in UML. It has three sections - Name, Attributes, and Operations. You can show as many or as few of the Attributes and Operations in the diagram. Most of the times for the sake of clarity the Attribute and Operation lists are suppressed.

You start from real world objects - abstract out what you do not care and go through the process of classification of what you care. A Class is the result of this classification. Classes are then used as templates within a software system to create software objects.

What are the various Relationships?

Dependency

Association

Aggregation

Composition

Generalization

Realization

Dependency is a semantic relationship between two things in which a change to one thing (independent thing) may affect the semantics of the other (dependent thing). This is a non-structural, "uses" type relationship as in Class Window uses Class Event. A change in Event (say a mouse click or key stroke) causes a change in Window but not vice versa. In UML Dependency relationship is denoted by a dashed line.

Association is a structural relationship between objects. The association may have a name. Professor "works for" University. Each end of the association has a role - Professor (employee) and University (employer). An Association is represented by a solid line in UML.

Multiplicity defines how many objects participate in a relationship. For each role you can specify the multiplicity of its class - how many objects of the class can be associated with an object of the other class.

Aggregation is a relationship between a whole and its parts. An object physically contains other objects - Car is physically composed of an Engine and four Wheels. A Family is a collection of Parents and Children. It is represented in UML as a solid line with an open diamond.

Composition is a form of aggregation with strong ownership and coincident lifetimes of the part with the aggregate. The whole "owns" the part and is responsible for the creation and destruction of the part. Student and schedule. It is represented as a solid line with a filled diamond in UML.

Generalization is a relationship among classes where one class shares the structure and behavior of one or more classes. Generalization defines a hierarchy of abstractions in which a subclass inherits from one or more superclass. Generalization is an "is-a-kind of" relationship.

A subclass inherits its parent's attributes, operations, and relationships. A subclass may add additional attributes, operations, relationships. It may also redefine inherited operations (use caution here!). This relationship is represented as an open arrow going from the subclass to its parent(s) in UML.

A subclass may be used anywhere the superclass is used, but not vice versa. Generalization is the name of the relationship. Inheritance is the mechanism that the generalization relationship represents.

Realization is the relationship between one class that serves as the contract that the other class agrees to carry out. (The interface and the class that implements the interface). It is represented as a dashed line and an arrow in UML.

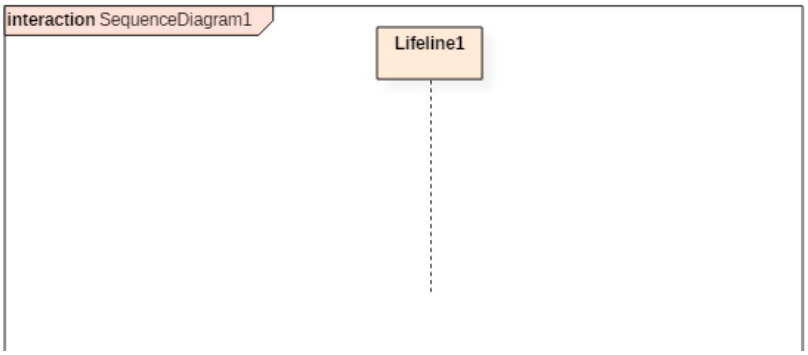
The sum total of information carried by the attributes of an object is called its **state**. If we know the state of an object, then we know everything there is to know about it. This last statement is true by definition. If in describing an object the designer finds that its attributes are not sufficient to specify its state completely, then they have overlooked some attributes.

If the state of an object is given by the values of all its attributes, then the state of the system if given by the states of all its objects. In other words, the values of all the attributes of all the objects completely specify the state of the system at any given time.

If state is so important, why can it not be represented on a class model? The reason is that there are far better ways to represent changes in state. The UML provides state transition modelling for this purpose. In a full design it may be necessary to provide state transition diagrams for some or all classes, to describe their state changes in detail. The pattern of state change over time is called the **dynamic behaviour** of a system, and modelling this is called **dynamic modelling**. As well as state transition modelling, the UML provides object interaction diagrams, activity diagrams, sequence diagrams and communication diagrams for dynamic modelling. One can, of course, describe the dynamic behaviour of classes in plain language as well, and one probably should.

INTERACTION DIAGRAM

Interaction Diagram are used in UML to establish communication between objects. It does not manipulate the data associated with the particular communication path. Interaction diagrams mostly focus on message passing and how these messages make up one functionality of a system. Interaction diagrams are designed to display how the objects will realize the particular requirements of a system. The critical component in an interaction diagram is lifeline and messages.
Various UML elements typically own interaction diagrams. The details of interaction can be shown using several notations such as sequence diagram, timing diagram, communication/collaboration diagram. Interaction diagrams capture the dynamic behavior of any system.



Notation of an Interaction Diagram

Following are the different types of interaction diagrams defined in UML:

- Sequence diagram
- Collaboration diagram
- Timing diagram

In UML, the interaction diagrams are used for the following purposes:

- Interaction diagrams are used to observe the dynamic behavior of a system.
- Interaction diagram visualizes the communication and sequence of message passing in the system.
- Interaction Modelling diagram represents the structural aspects of various objects in the system.
- Interaction diagram represents the ordered sequence of interactions within a system.
- Interaction diagram provides the means of visualizing the real time data via UML.
- UML Interaction Diagrams can be used to explain the architecture of an object-oriented or a distributed system.

Important terminology

An interaction diagram contains lifelines, messages, operators, state invariants and constraints.

Lifeline

A lifeline represents a single participant in an interaction. It describes how an instance of a specific classifier participates in the interaction.

A lifeline represents a role that an instance of the classifier may play in the interaction. Following are various attributes of a lifeline,

1. **Name**
1. It is used to refer the lifeline within a specific interaction.
2. A name of a lifeline is optional.
2. **Type**
1. It is the name of a classifier of which the lifeline represents an instance.
3. **Selector**
1. It is a Boolean condition which is used to select a particular instance that satisfies the requirement.
2. Selector attribute is also optional.

The notation of lifeline is explained in the notation section.

Messages

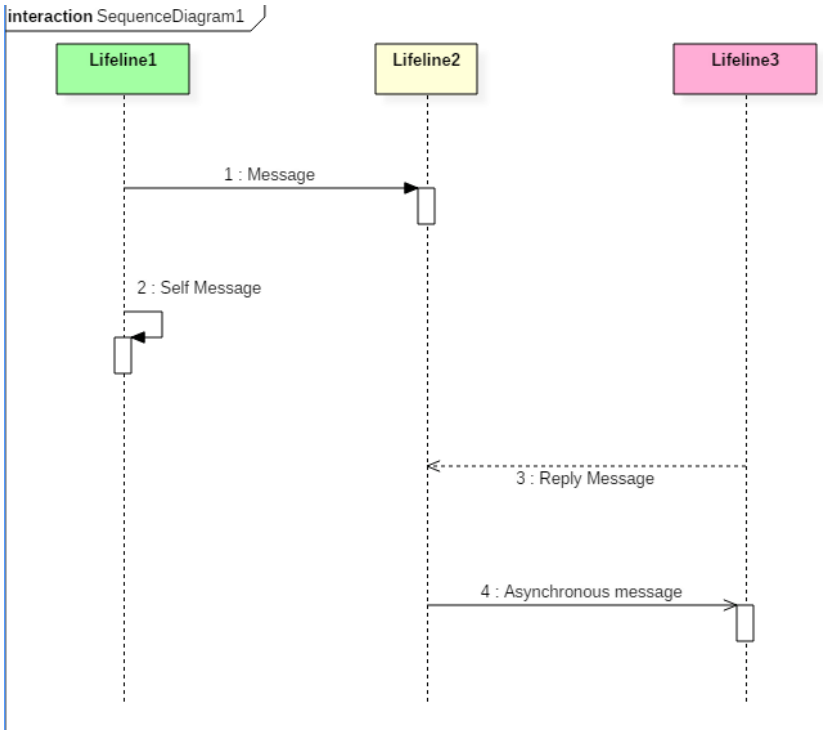
A message is a specific type of communication between two lifelines in an interaction. A message involves following activities,

1. A call message which is used to call an operation.
2. A message to create an instance.
3. A message to destroy an instance.
4. For sending a signal.

SEQUENCE DIAGRAM

A **Sequence Diagram** simply depicts interaction between objects in a sequential order. The purpose of a sequence diagram in UML is to visualize the sequence of a message flow in the system. The sequence diagram shows the interaction between two lifelines as a time-ordered sequence of events.

- A sequence diagram shows an implementation of a scenario in the system. Lifelines in the system take part during the execution of a system.
- In a sequence diagram, a lifeline is represented by a vertical bar.
- A message flow between two or more objects is represented using a vertical dotted line which extends across the bottom of the page.
- In a sequence diagram, different types of messages and operators are used which are described above.
- In a sequence diagram, iteration and branching are also used.



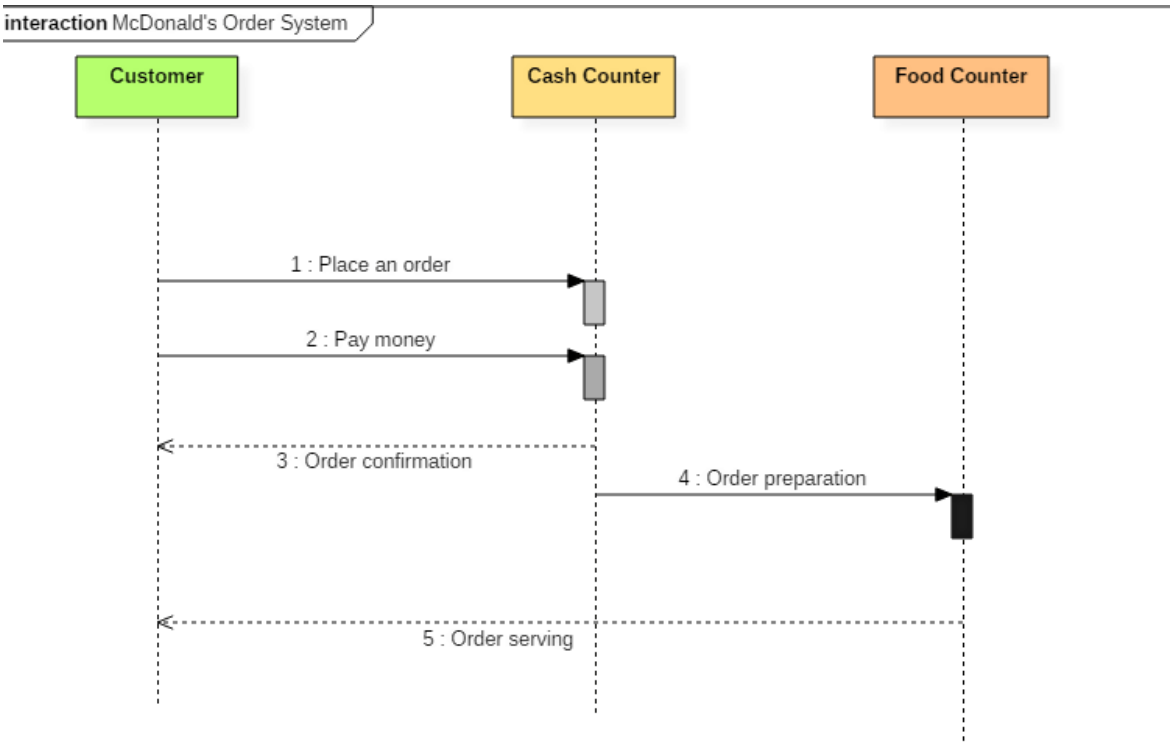
Notations in Sequence Diagram

The above sequence diagram contains lifeline notations and notation of various messages used in a sequence diagram such as a create, reply, asynchronous message, etc.

Sequence diagram example

The following sequence diagram example represents McDonald's ordering system:

Sequence diagram of Mcdonald's ordering system



The ordered sequence of events in a given sequence diagram is as follows:

1. Place an order.
2. Pay money to the cash counter.
3. Order Confirmation.
4. Order preparation.
5. Order serving.

If one changes the order of the operations, then it may result in crashing the program. It can also lead to generating incorrect or buggy results. Each sequence in the above-given sequence diagram is denoted using a different type of message. One cannot use the same type of message to denote all the interactions in the diagram because it creates complications in the system.

Benefits of a Sequence Diagram

- Sequence diagrams are used to explore any real application or a system.
- Sequence diagrams are used to represent message flow from one object to another object.

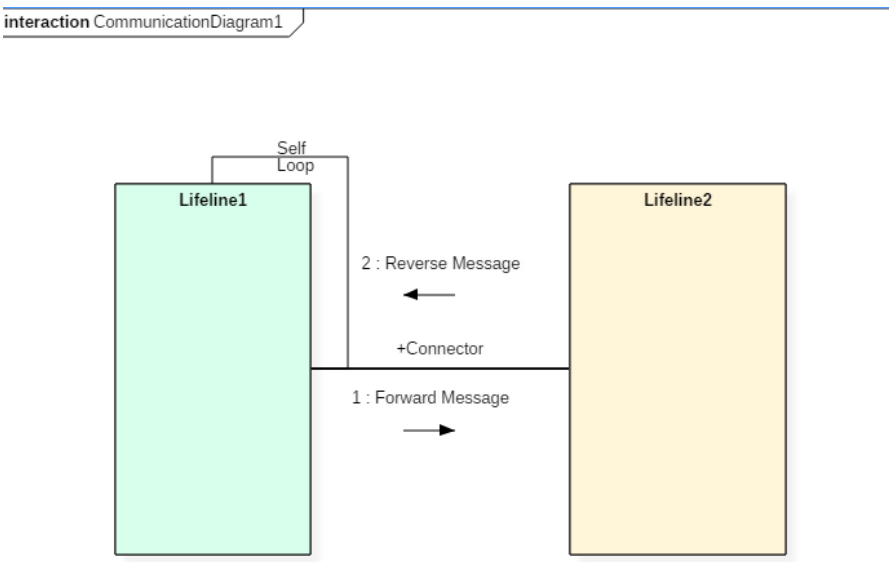
- Sequence diagrams are easier to maintain.
- Sequence diagrams are easier to generate.
- Sequence diagrams can be easily updated according to the changes within a system.
- Sequence diagram allows reverse as well as forward engineering.

COLLABORATION DIAGRAM

Collaboration Diagram depicts the relationships and interactions among software objects. They are used to understand the object architecture within a system rather than the flow of a message as in a sequence diagram. They are also known as "Communication Diagrams."
As per Object-Oriented Programming (OOPs), an object entity has various attributes associated with it. Usually, there are multiple objects present inside an object-oriented system where each object can be associated with any other object inside the system. Collaboration Diagrams are used to explore the architecture of objects inside the system. The message flow between the objects can be represented using a collaboration diagram.

Benefits of Collaboration Diagram

- It is also called as a communication diagram.
- It emphasizes the structural aspects of an interaction diagram – how lifeline connects.
- Its syntax is similar to that of sequence diagram except that lifeline don't have tails.
- Messages passed over sequencing is indicated by numbering each message hierarchically.
- Compared to the sequence diagram communication diagram is semantically weak.
- Object diagrams are special case of communication diagram.
- It allows you to focus on the elements rather than focusing on the message flow as described in the sequence diagram.
- Sequence diagrams can be easily converted into a collaboration diagram as collaboration diagrams are not very expressive.
- While modeling collaboration diagrams w.r.t sequence diagrams, some information may be lost.



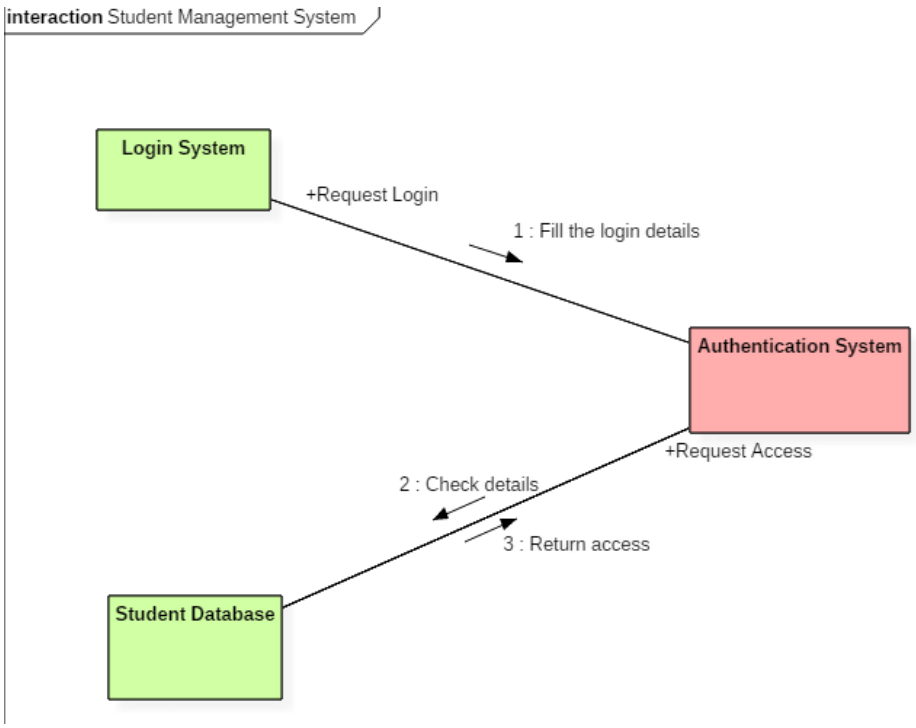
The above collaboration diagram notation contains lifelines along with connectors, self-loops, forward, and reverse messages used in a collaboration diagram.

Drawbacks of a Collaboration Diagram

- Collaboration diagrams can become complex when too many objects are present within the system.
- It is hard to explore each object inside the system.
- Collaboration diagrams are time consuming.
- The object is destroyed after the termination of a program.
- The state of an object changes momentarily, which makes it difficult to keep track of every single change the occurs within an object of a system.

Collaboration diagram Example

Following diagram represents the sequencing over student management system:



Collaboration diagram for student management system

The above collaboration diagram represents a student information management system. The flow of communication in the above diagram is given by,

1. A student requests a login through the login system.
2. An authentication mechanism of software checks the request.

3. If a student entry exists in the database, then the access is allowed; otherwise, an error is returned.

Introduction

The Unified Modeling Language is a modeling language that can be used for any purpose. The main goal of UML is to establish a standard for visualizing the design of the system. It looks a lot like designs in other branches of engineering.

Unified Modeling Language is a visual language rather than a programming language. Unified Modeling Language diagrams are used to depict a system's behavior and structure. UML is a modeling, design, and analysis tool for software engineers, entrepreneurs, business people and system architects. UML was approved as a standard by the OMG(Object Management Group) in 1997. Since then, the OMG has been in charge of it. In 2005, the ISO(International Organization for Standardization) accepted UML as a standard. Unified Modeling Language has been updated throughout time and is examined on a regular basis.

The Need for UML

- Complex applications involve the collaboration and planning of various teams, requiring a clear and straightforward method of communication between them.
- Code is not understood by people in businesses. As a result, Unified Modeling Language becomes critical for communicating the system's core requirements, features, and procedures to non-programmers.
- When teams can view processes, user interactions, and the system's static structure, they save a lot of time in the long run.

UML Architecture

Software architecture refers to how a software system is constructed at its most fundamental level. It is necessary to think big from a variety of angles while keeping quality and design in mind.

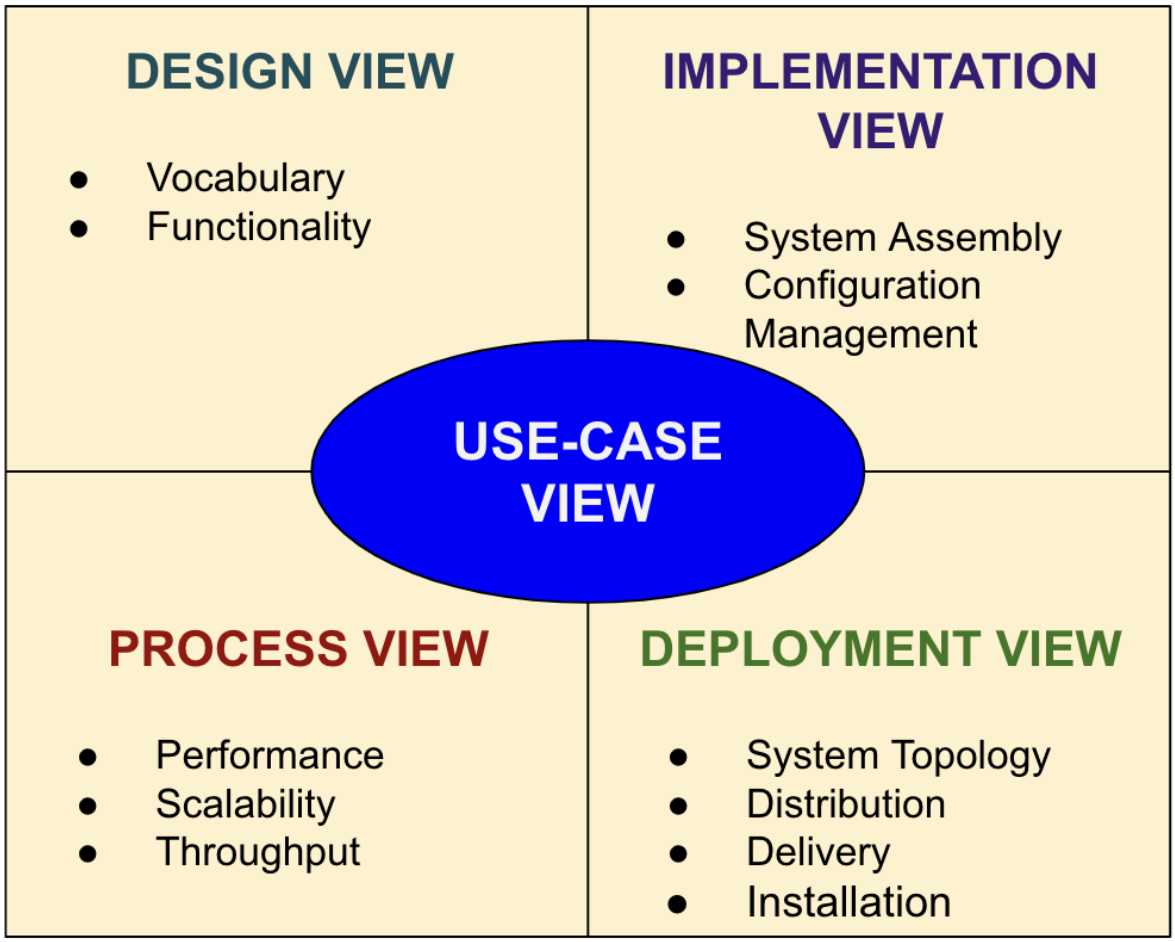
The software team is responsible for a variety of practical concerns, including:

- The development team's structure.
- The company's requirements.
- The development cycle.
- The structure's original intent.

The basic design of a comprehensive software system is provided by software architecture. It specifies the parts that make up the system, their functions, and how they interact with one another. In a nutshell, it's a large picture or overall structure of the entire system, showing how everything interacts.

Software developers, project managers, clients, and end-users can all benefit from the software architecture. Each will bring distinct agendas to a project and will have various perspectives on the system. It also includes a compilation of different perspectives. It's best described as a compilation of five points of view:

- Use-Case view
- Design view
- Implementation view
- Process view
- Development view



Building Blocks in UML

Things, relationships, and diagrams are the three main building blocks of UML. By rotating multiple different blocks, building blocks create one full UML model diagram. It is crucial in the creation of UML diagrams.

The following are the basic UML building blocks:

- Things
- Relationships
- Diagrams

Things

Things refer to anything that is a physical entity or object. It can be broken down into various categories:

- Structural Things
- Behavioral Things
- Grouping Things
- Annotational Things

Relationships

It shows how things are connected in meaningful ways. It depicts the relationships between entities and defines an application's functionality. Relationships can be divided into four categories:

- Dependency
- Association
- Generalization
- Realization

Diagrams

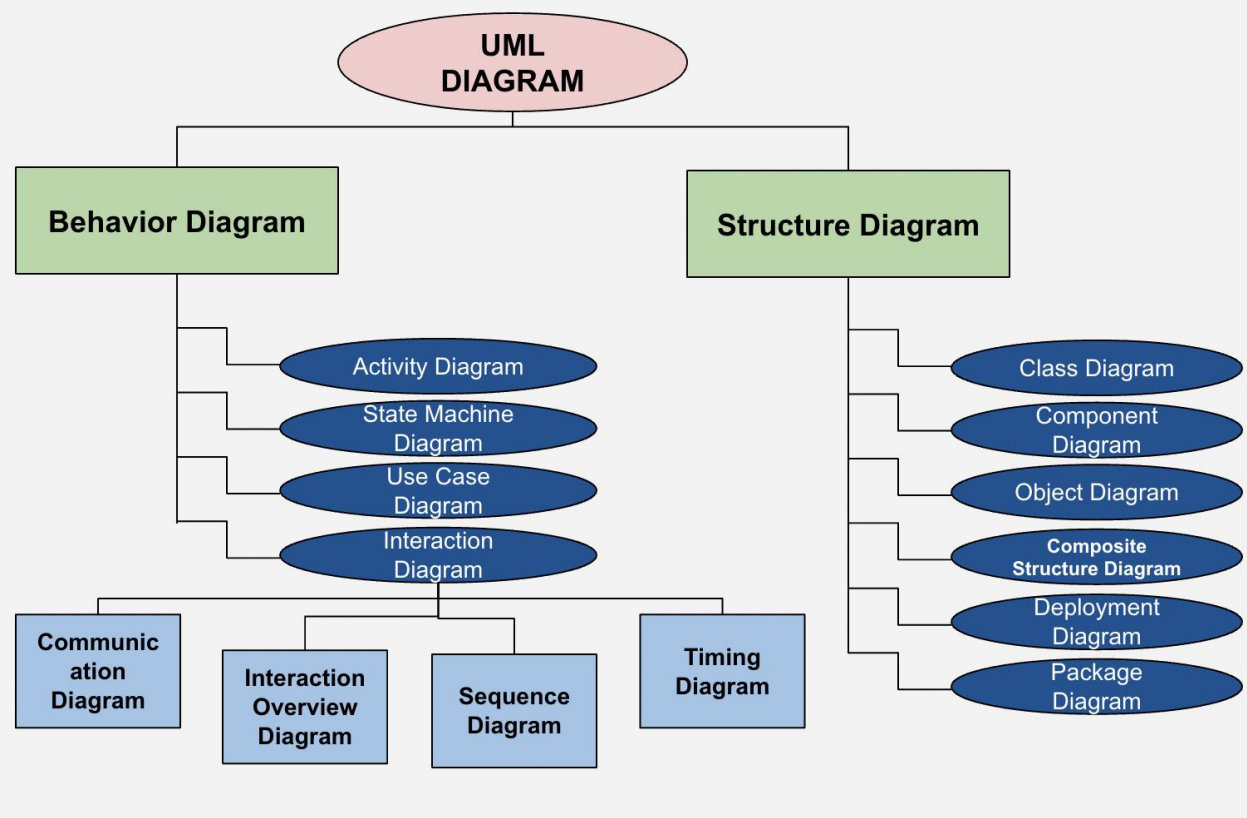
The diagrams are the visual representations of the models, which include symbols and text. In the context of the UML diagram, each symbol has a different meaning.

UML diagrams can be divided into three groups, as shown below:

- Structural Diagram
- Behavioral Diagram
- Interaction Diagram

UML Diagrams

Object-oriented design and analysis are linked to Unified Modeling Language. Structural, behavioral, and interaction diagrams are the three types of UML diagrams. In the diagram below, the representations are arranged in a hierarchical order:



UML creates diagrams by combining elements and forming associations between them.

UML diagrams are divided into three categories:

Structural Diagrams

It depicts a system's structure and represents a system's static view. It displays several objects that can be found in the system.

The structural schematics follow:

- Component Diagrams
- Object Diagrams
- Class Diagrams
- Composite Structure Diagrams
- Package Diagrams
- Deployment Diagrams

Behavioral Diagrams

It depicts a system's behavioral characteristics. It is concerned with the system's dynamic components. The following diagrams are included:

- Use Case Diagrams
- State Machine Diagrams
- Activity Diagrams

Interaction Diagrams

It's a subset of behavioral diagrams. It displays the data flow between two objects and their interaction. The following are some UML interaction diagrams:

- Sequence Diagrams
- Timing Diagrams
- Communication Diagrams
- Interaction Overview Diagrams

Use of Object-Oriented Concepts in UML

UML is the next generation of object-oriented(OO) analysis and design.

Data and methods that control the data are both contained in an **object**. The data represents the object's current status. A **class** is a type of object with a hierarchy to model a real-world system. The hierarchy is represented by inheritance, and the classes can be linked in various ways depending on the situation.

Objects are real-world entities that occur all around us, and UML may be used to express basic principles like **abstraction**, **encapsulation**, **inheritance**, and **polymorphism**.

UML can represent all of the concepts found in object-oriented analysis and design. Only object-oriented notions are represented in UML diagrams. As a result, it is very essential to understand the object-oriented concept before learning UML.

Some basic concepts in the object-oriented world are:

Class - A class is a blueprint for an object, defining its structure and functionality.

Objects - Objects aid in the decomposition and modularization of huge systems. Modularity allows us to break down our system into easily understandable components, allowing us to create it piece by piece. An object is a system's core unit (building block) that represents an entity.

Inheritance - It is the mechanism that allows child classes to inherit properties from their parent classes.

Abstraction - It is the mechanism for hiding implementation details from the user.

Encapsulation - It is the process of tying data together and shielding it from the outside world.

Polymorphism - It is the mechanism that allows functions or things to exist in several forms.

Some of Uml tools:

1) Adobe Spark

2) Umbrello

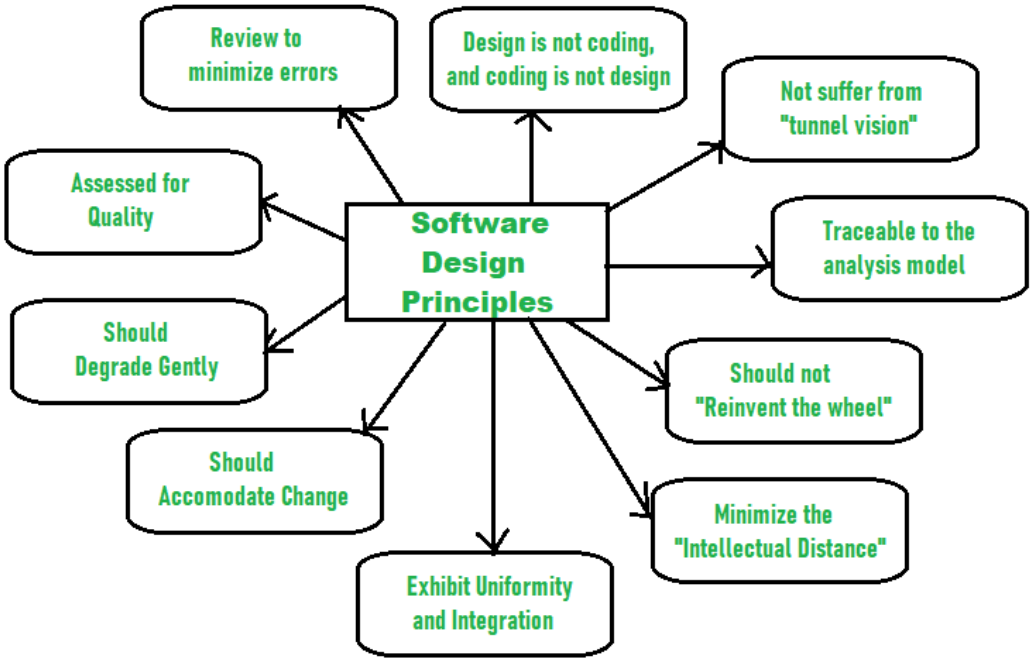
4) Lucidchart

5) WhiteStarUML

6) Visio

5) StarUML

UNIT -3: PRINCIPLES OF SOFTWARE DESIGN



Design means to draw or plan something to show the look, functions and working of it.

Software Design is also a process to plan or convert the software requirements into a step that are needed to be carried out to develop a software system. There are several principles that are used to organize and arrange the structural components of Software design. Software Designs in which these principles are applied affect the content and the working process of the software from the beginning.
These principles are stated below :

Principles Of Software Design :

1. **Should not suffer from "Tunnel Vision" –**
   While designing the process, it should not suffer from "tunnel vision" which means that is should not only focus on completing or achieving the aim but on other effects also.

2. **Traceable to analysis model –**
   The design process should be traceable to the analysis model which means it should satisfy all the requirements that software requires to develop a high-quality product.

3. **Should not "Reinvent The Wheel" –**
   The design process should not reinvent the wheel that means it should not waste time or effort in creating things that already exist. Due to this, the overall development will get increased.

4. **Minimize Intellectual distance –**
   The design process should reduce the gap between real-world problems and software solutions for that problem meaning it should simply minimize intellectual distance.

5. **Exhibit uniformity and integration –**
   The design should display uniformity which means it should be uniform throughout the process without any change. Integration means it should mix or combine all parts of software i.e. subsystems into one system.

6. **Accommodate change –**
   The software should be designed in such a way that it accommodates the change implying that the software should adjust to the change that is required to be done as per the user's need.

7. **Degrade gently –**
   The software should be designed in such a way that it degrades gracefully which means it should work properly even if an error occurs during the execution.

8. **Assessed or quality –**
   The design should be assessed or evaluated for the quality meaning that during the evaluation, the quality of the design needs to be checked and focused on.

9. **Review to discover errors –**
   The design should be reviewed which means that the overall evaluation should be done to check if there is any error present or if it can be minimized.

10. **Design is not coding and coding is not design –**
    Design means describing the logic of the program to solve any problem and coding is a type of language that is used for the implementation of a design.

What is Gang of Four (GOF)?

In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.

These authors are collectively known as **Gang of Four (GOF)**. According to these authors design patterns are primarily based on the following principles of object orientated design.

- Program to an interface not an implementation
- Favor object composition over inheritance

DESIGN PATTERNS

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Uses of Design Patterns

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

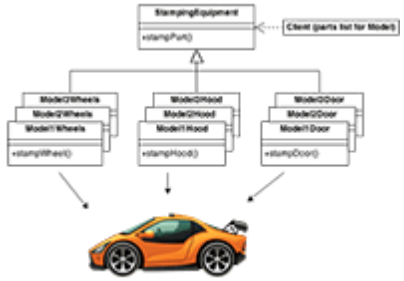
Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

Creational design patterns

These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use

inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.



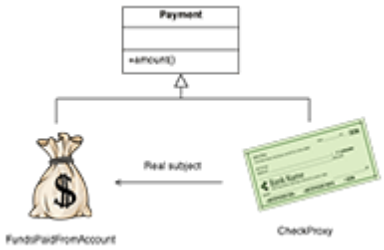
- **Abstract Factory**
  Creates an instance of several families of classes
- **Builder**
  Separates object construction from its representation
- **Factory Method**
  Creates an instance of several derived classes
- **Object Pool**
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
- **Prototype**
  A fully initialized instance to be copied or cloned
- **Singleton**
  A class of which only a single instance can exist

Structural design patterns

These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

- **Adapter**
  Match interfaces of different classes

- **Bridge**
  Separates an object's interface from its implementation
- **Composite**
  A tree structure of simple and composite objects
- **Decorator**
  Add responsibilities to objects dynamically
- **Facade**
  A single class that represents an entire subsystem
- **Flyweight**
  A fine-grained instance used for efficient sharing

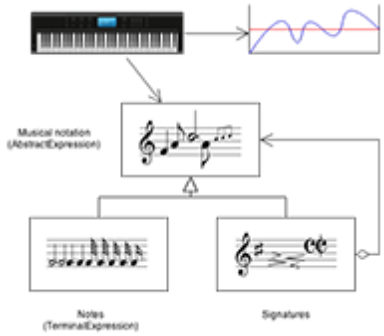


- **Private Class Data**
  Restricts accessor/mutator access

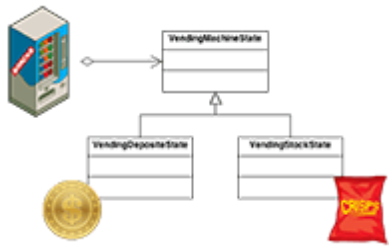
- **Proxy**
  An object representing another object

Behavioral design patterns

These design patterns are all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.



- **Chain of responsibility**
  A way of passing a request between a chain of objects
- **Command**
  Encapsulate a command request as an object
- **Interpreter**
  A way to include language elements in a program

- **Iterator**
  Sequentially access the elements of a collection
- **Mediator**
  Defines simplified communication between classes
- **Memento**
  Capture and restore an object's internal state
- **Null Object**
  Designed to act as a default value of an object
- **Observer**
  A way of notifying change to a number of classes
- 

  State
  Alter an object's behavior when its state changes

- **Strategy**
  Encapsulates an algorithm inside a class
- **Template method**
  Defer the exact steps of an algorithm to a subclass
- **Visitor**
  Defines a new operation to a class without change

Criticism

The concept of design patterns has been criticized by some in the field of computer science.

Creational	Structural	Behavioral
• Factory Method	• Adapter	• Interperter
• Abstract Factory • Builder • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Momento • Observer • State • Strategy • Visitor

GRASP DESIGN PRINCIPLES

GRASP design Principles, GRASP stands for General Responsibility Assignment Software Patterns. It guides in assigning responsibilities to collaborate objects.

9 GRASP Design patterns

Following are the list of GRASP desgin Principles which I try to explain in simple way and concise way.

- Creator
- Information Expert
- Low Coupling
- Controller
- High Cohesion
- Indirection
- Polymorphism
- Protected Variations
- Pure Fabrication

How to Attache Responsiblity to a Object

Responsibility can be accomplished by a single object or a group of object collaboratively accomplish a responsibility.

GRASP helps us in deciding which responsibility should be assigned to which object/class.

Identify the objects and responsibilities from the problem domain, and also identify how objects interact with each other.

Define blue print for those objects e.g. class with methods implementing those responsibilities.

The following are mail design principle

1. **Creator of GRASP design Patterns**

Who creates an Object? Or who should create a new instance of some class?

"Container" object creates "contained" objects.

Decide who can be creator based on the objects association and their interaction.

2. **Expert**

provided an object obj, which responsibilities can be assigned to obj?

Expert principle says that assign those responsibilities to obj for which obj has the information to fulfill that responsibility.

They have all the information needed to perform operations, or in some cases they collaborate with others to fulfill their responsibilities.

3. **Low Coupling**

How strongly the objects are connected to each other?

Coupling – object depending on other object.

When depended upon element changes, it affects the dependant also.

Low Coupling – How can we reduce the impact of change in depended upon elements on dependant elements.

Prefer low coupling – assign responsibilities so that coupling remain low.

Minimizes the dependency hence making system maintainable, efficient and code reusable

Two elements can be coupled, by following if

One element has aggregation/composition or association with another element.

– One element implements/extends other element.

4. **High Cohesion**

How are the operations of any element are functionally related?

Related responsibilities in to one manageable unit.

Prefer high cohesion

Clearly defines the purpose of the element

Benefits

– Easily understandable and maintainable.

– Code reuse

– Low coupling

5. **Controller in GRASP design principles**

Deals with how to delegate the request from the UI layer objects to domain layer objects.

when a request comes from UI layer object, Controller pattern helps us in determining what is that first object that receive the message from the UI layer objects.

This object is called controller object which receives request from UI layer object and then controls/coordinates with other object of the domain layer to fulfill the request.

It delegates the work to other class and coordinates the overall activity.

We can make an object as Controller, if

Object represents the overall system (facade controller)

Object represent a use case, handling a sequence of operations

(session controller).

Benefits of Controller:

– can reuse this controller class.

– Can use to maintain the state of the use case.

– Can control the sequence of the activities

is Bloated Controllers Good Or Bad

Controller class is called bloated, if the class is overloaded with too many responsibilities.

follwoing are the solution

Add more controllers

Controller class also performing many tasks instead of delegating to other class.

controller class has to delegate things to others.

6. **Polymorphism**

How to handle related but varying elements based on element type?

Polymorphism guides us in deciding which object is responsible for handling those varying elements.

Benefits: handling new variations will become easy.

7. **Pure Fabrication**

Fabricated class/ artificial class – assign set of related responsibilities that doesn't represent any domain object.

Provides a highly cohesive set of activities.

Behavioral decomposed – implements some algorithm.

Examples: Adapter, Strategy

Benefits: High cohesion, low coupling and can reuse this class.

8. **Indirection**

How can we avoid a direct coupling between two or more elements.

Indirection introduces an intermediate unit to communicate between the other units, so that the other units are not directly coupled.

Benefits: low coupling, e.g Facade, Adapter, Obserever.

Class Employee provides a level of indirection to other units of the system.

9. **Protected Variation**

This section is about P: protoected Variation from GRASP design Principles.

How to avoid impact of variations of some elements on the other elements.

It provides a well defined interface so that the there will be no affect on other units.

Provides flexibility and protection from variations.

Provides more structured design. Example: polymorphism, data encapsulation, interfaces

OMT METHODOLOGY

Four phases of OMT (can be performed iteratively)

- Analysis: Objects, dynamic and functional models
- System Design: Basic architecture of the system.
- Object Design: Static, dynamic and functional models of objects.
- Implementation: Reusable, extendible and robust code.

Three different parts of OMT Modeling

Object model

• Represents the static, structural, 'data' aspects of a system

Dynamic model

• Represents the temporal, behavioural,'control' aspects of a system

Functional model

• Represents the transformational,'functional' aspects of a system

OBJECT MODEL

The object model describes:

• the structure of the object

• the relationship of one object with other objects

• attributes and operations of the objects.

- Captures the concepts from the real world that are useful for the application.
- Represented by class diagram and object diagram.

OBJECT DIAGRAM

Models the instances of classes that are present in class diagram.

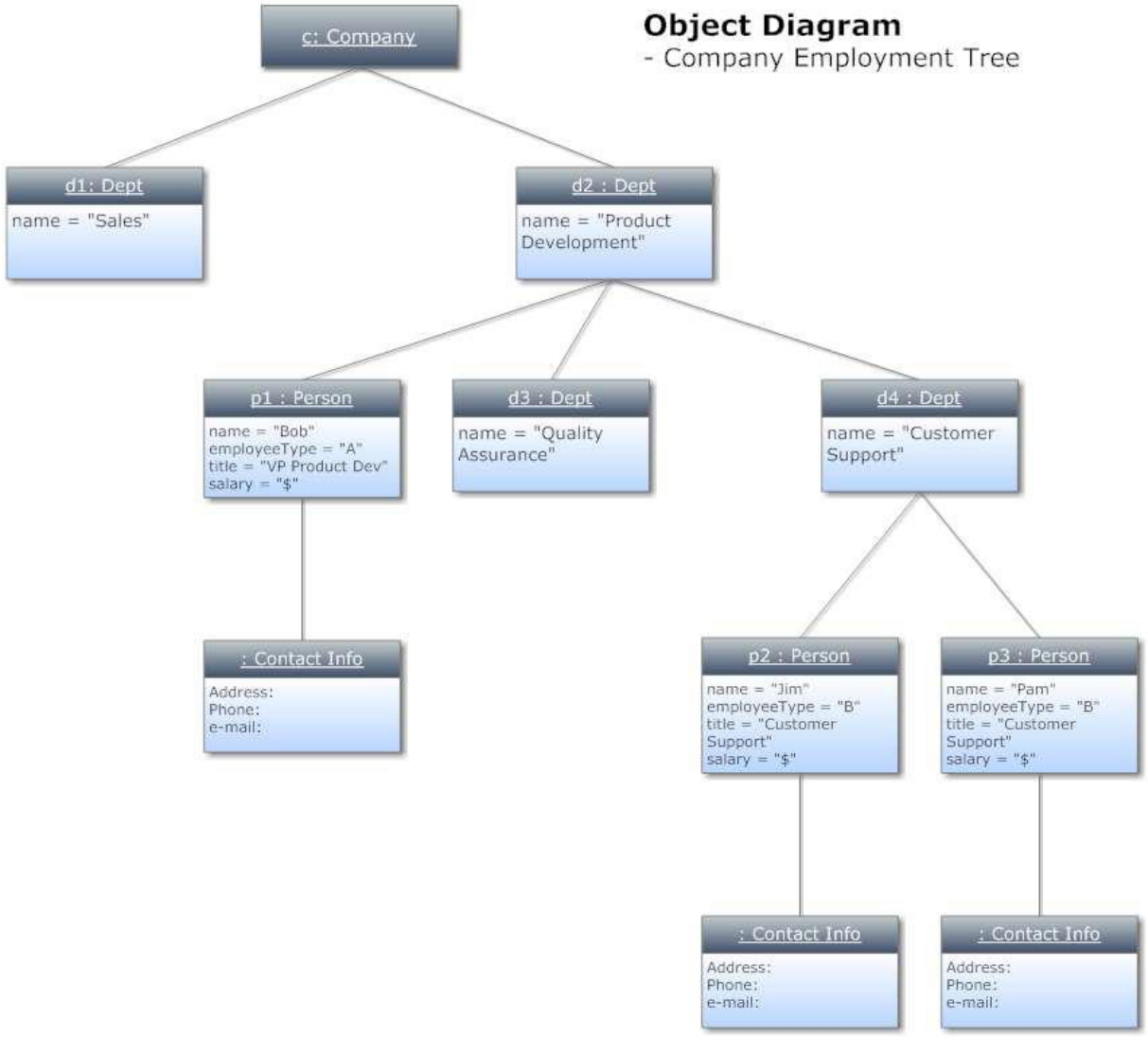
Used to model the static design view of the system.

Defines the attributes and operations of each object.

Object diagram contains:

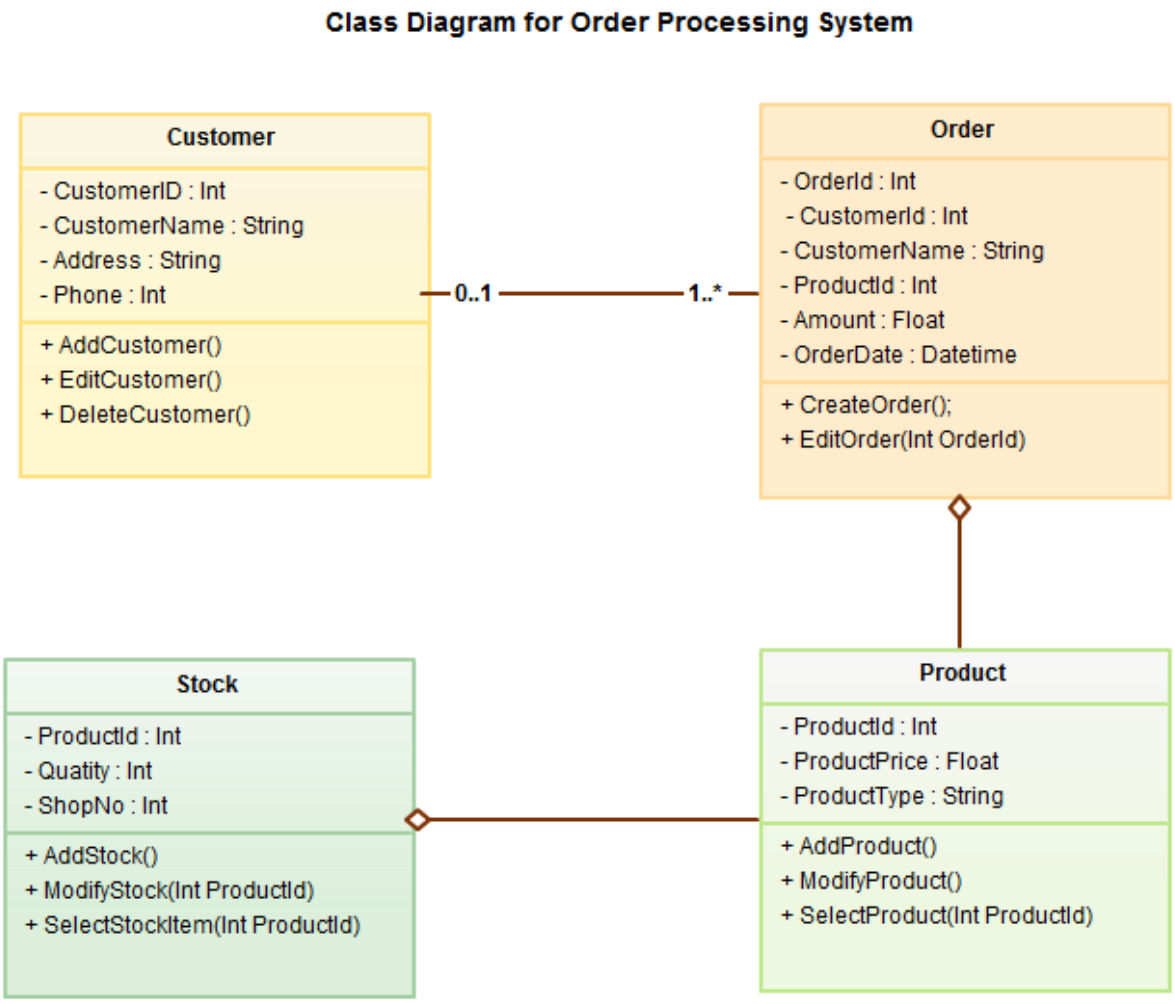
• Objects

• Links



CLASS DIAGRAM

- A graphical representation used for modeling classes and their relationships.
- Describes all possible objects belonging to the classes.
- Used for abstract modeling and for implementing actual program
- The class diagram is concise and can be understood easily.
- Classes are interconnected by association lines.



OMT Dynamic Model

- States, transitions, events and actions.
- Concerned with the time and sequencing of the operations of the object.
- Captures control aspect of the system.
- Represented by state transition diagram.

STATE TRANSITION DIAGRAM (1)

- State:

Some behavior of a system that is observable and that lasts for some period of time.

- A state is when a system is:

• Doing something – e.g., heating oven, mixing ingredients, accelerating engine,

• Waiting for something to happen –Waiting for user to enter password, waiting for sensor reading.

Transition:

(Virtually) instantaneous change in state (behavior).

STATE TRANSITION DIAGRAM (2)

A condition is typically some kind of event, e.g.:

• Signal

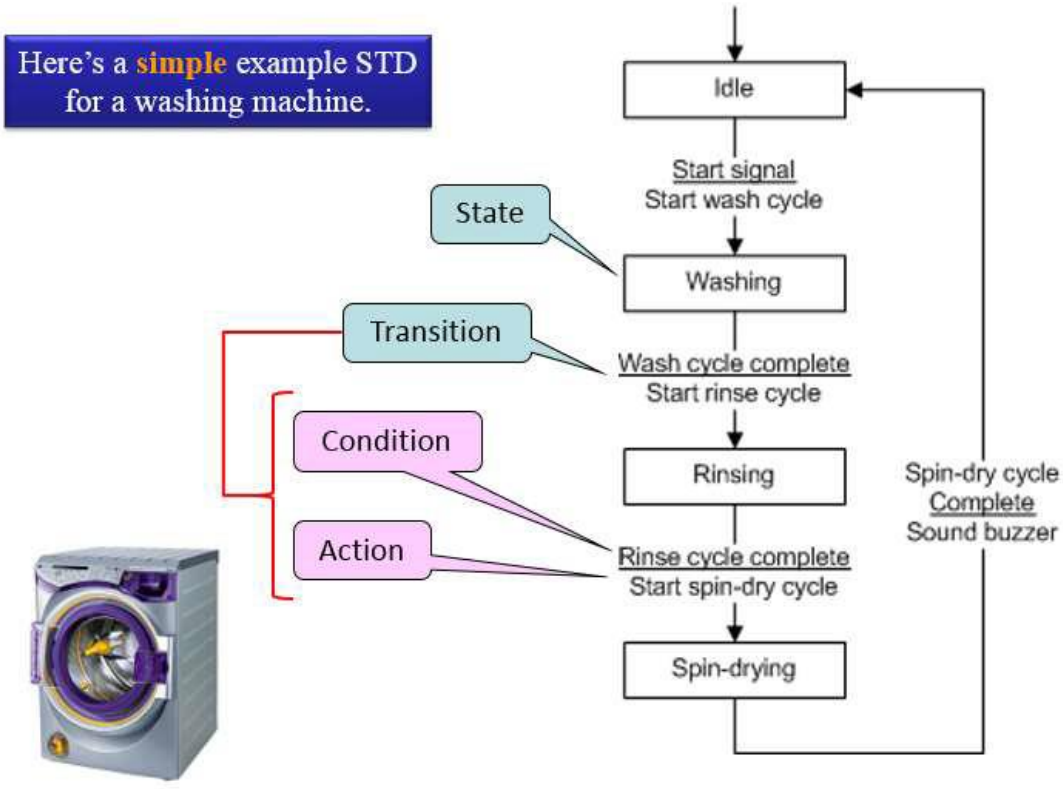
• Arrival of an object (data/material)

An action is the appropriate output or response to the event, e.g.:

• Signal or message

• Transfer of an object,

• Calculation



Object model and dynamic model

 The dynamic model describes the control structure of objects.

 The states of the dynamic model can be related to classes of attribute and links to values of an object.

 Events and actions can be represented as operations on the object model.

 The object model concepts of generalization, aggregation and inheritance also apply to the dynamic model.

Object model and dynamic model

 The dynamic model describes the control structure of objects.

 The states of the dynamic model can be related to classes of attribute and links to values of an object.

 Events and actions can be represented as operations on the object model.

 The object model concepts of generalization, aggregation and inheritance also apply to the dynamic model.

UNIT -4 SOFTWARE TESTING

Software Testing is a method to check whether the actual software product matches expected requirements and to ensure that software product is Defect free. It involves execution of software/system components using manual or automated tools to evaluate one or more properties of interest. The purpose of software testing is to identify errors, gaps or missing requirements in contrast to actual requirements.

Some prefer saying Software testing definition as a White Box and Black Box Testing. In simple terms, Software Testing means the Verification of Application Under Test (AUT). This Software Testing course introduces testing software to the audience and justifies the importance of software testing.

Why Software Testing is Important?

Software Testing is Important because if there are any bugs or errors in the software, it can be identified early and can be solved before delivery of the software product. Properly tested software product ensures reliability, security and high performance which further results in time saving, cost effectiveness and customer satisfaction.

What is the need of Testing?

Testing is important because software bugs could be expensive or even dangerous. Software bugs can potentially cause monetary and human loss, and history is full of such examples.

- In April 2015, Bloomberg terminal in London crashed due to software glitch affected more than 300,000 traders on financial markets. It forced the government to postpone a 3bn pound debt sale.
- Nissan cars recalled over 1 million cars from the market due to software failure in the airbag sensory detectors. There has been reported two accident due to this software failure.
- Starbucks was forced to close about 60 percent of stores in the U.S and Canada due to software failure in its POS system. At one point, the store served coffee for free as they were unable to process the transaction.
- Some of Amazon's third-party retailers saw their product price is reduced to 1p due to a software glitch. They were left with heavy losses.
- Vulnerability in Windows 10. This bug enables users to escape from security sandboxes through a flaw in the win32k system.
- In 2015 fighter plane F-35 fell victim to a software bug, making it unable to detect targets correctly.
- China Airlines Airbus A300 crashed due to a software bug on April 26, 1994, killing 264 innocents live
- In 1985, Canada's Therac-25 radiation therapy machine malfunctioned due to software bug and delivered lethal radiation doses to patients, leaving 3 people dead and critically injuring 3 others.
- In April of 1999, a software bug caused the failure of a $1.2 billion military satellite launch, the costliest accident in history
- In May of 1996, a software bug caused the bank accounts of 823 customers of a major U.S. bank to be credited with 920 million US dollars.

What are the benefits of Software Testing?

Here are the benefits of using software testing:

- **Cost-Effective:** It is one of the important advantages of software testing. Testing any IT project on time helps you to save your money for the long term. In case if the bugs caught in the earlier stage of software testing, it costs less to fix.
- **Security:** It is the most vulnerable and sensitive benefit of software testing. People are looking for trusted products. It helps in removing risks and problems earlier.
- **Product quality:** It is an essential requirement of any software product. Testing ensures a quality product is delivered to customers.
- **Customer Satisfaction:** The main aim of any product is to give satisfaction to their customers. UI/UX Testing ensures the best user experience.

Types of Software Testing :Here are the software testing types:

Typically Testing is classified into three categories.

- Functional Testing
- Non-Functional Testing or Performance Testing
- Maintenance (Regression and Maintenance)

Testing Category	Types of Testing
Functional Testing	• Unit Testing • Integration Testing • Smoke • UAT (User Acceptance Testing) • Localization • Globalization • Interoperability • So on
Non-Functional Testing	• Performance • Endurance • Load • Volume • Scalability • Usability • So on
Maintenance	• Regression • Maintenance

Testing Strategies in Software Engineering

Here are important strategies in software engineering:

Unit Testing: This software testing basic approach is followed by the programmer to test the unit of the program. It helps developers to know whether the individual unit of the code is working properly or not.

Integration testing: It focuses on the construction and design of the software. You need to see that the integrated units are working without errors or not.

System testing: In this method, your software is compiled as a whole and then tested as a whole. This testing strategy checks the functionality, security, portability, amongst others.

Program Testing

Program Testing in software testing is a method of executing an actual software program with the aim of testing program behavior and finding errors. The software program is executed with test case data to analyse the program behavior or response to the test data. A good program testing is one which has high chances of finding bugs.

Types of Software Testing

Introduction:-

Testing is the process of executing a program with the aim of finding errors. To make our software perform well it should be error-free. If testing is done successfully it will remove all the errors from the software.

Principles of Testing:-

(i) All the test should meet the customer requirements
(ii) To make our software testing should be performed by a third party
(iii) Exhaustive testing is not possible. As we need the optimal amount of testing based on the risk assessment of the application.
(iv) All the test to be conducted should be planned before implementing it
(v) It follows the Pareto rule(80/20 rule) which states that 80% of errors come from 20% of program components.
(vi) Start testing with small parts and extend it to large parts.

Types of Testing:-

1. Unit Testing
It focuses on the smallest unit of software design. In this, we test an individual unit or group of interrelated units. It is often done by the programmer by using sample input and observing its corresponding outputs.
Example:

a) In a program we are checking if loop, method or

function is working fine

b) Misunderstood or incorrect, arithmetic precedence.

c) Incorrect initialization

2. Integration Testing
The objective is to take unit tested components and build a program structure that has been dictated by design. Integration testing is testing in which a group of components is combined to produce output.

Integration testing is of four types: (i) Top-down (ii) Bottom-up (iii) Sandwich (iv) Big-Bang Example

(a) Black Box testing:- It is used for validation.

In this we ignore internal working mechanism and focuse on **what is the output?**

(b) White Box testing:- It is used for verification. In this we focus on internal mechanism i.e.**how the output is achieved?**

3. Regression Testing

Every time a new module is added leads to changes in the program. This type of testing makes sure that the whole component works properly even after adding components to the complete program.
Example

In school record suppose we have module staff, students and finance combining these modules and checking if on integration these module works fine is regression testing

4. Smoke Testing

This test is done to make sure that software under testing is ready or stable for further testing It is called a smoke test as the testing an initial pass is done to check if it did not catch the fire or smoke in the initial switch on.
Example:

If project has 2 modules so before going to module make sure that module 1 works properly

5. Alpha Testing

This is a type of validation testing. It is a type of *acceptance testing* which is done before the product is released to customers. It is typically done by QA people.
Example:
When software testing is performed internally within the organization

6. Beta Testing

The beta test is conducted at one or more customer sites by the end-user of the software. This version is released for a limited number of users for testing in a real-time environment
Example:

When software testing is performed for the limited number of people

7. System Testing

This software is tested such that it works fine for the different operating systems. It is covered under the black box testing technique. In this, we just focus on the required input and output without focusing on internal working.
In this, we have security testing, recovery testing, stress testing, and performance testing
Example:

This include functional as well as non functional testing

8. Stress Testing

In this, we give unfavorable conditions to the system and check how they perform in those conditions.
Example:

(a) Test cases that require maximum memory or other resources are executed

(b) Test cases that may cause thrashing in a virtual operating system

(c) Test cases that may cause excessive disk requirement

9. Performance Testing

It is designed to test the run-time performance of software within the context of an integrated system. It is used to test the speed and effectiveness of the program. It is also called load testing. In it we check, what is the performance of the system in the given load.
Example:

Checking number of processor cycles.

10. Object-Oriented Testing

This testing is a combination of various testing techniques that help to verify and validate object-oriented software. This testing is done in the following manner:

- Testing of Requirements,
- Design and Analysis of Testing,
- Testing of Code,
- Integration testing,
- System testing,
- User Testing.

Software Testing | Basics

Software testing can be stated as the process of verifying and validating that software or application is bug-free, meets the technical requirements as guided by its 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 the 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 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 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 the 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 a 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 the 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 concern 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, has access to its 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 a must.
Also known as closed box/data-driven testing.	Also known as clear box/structural testing.

Black Box Testing	White Box Testing
End users, testers, and developers.	Normally done by testers and developers.
This can only be done by a 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.

Note: Software testing is a very broad and vast topic and is considered to be an integral and very important part of software development and hence should be given its due importance.

Verification Methods in Software Verification

What is software verification?
Reviewing of any software for the purpose of finding faults is known as software verification. Verification is the process of checking that a software achieves its goal without any bugs. It is the process to ensure whether the product that is developed is right or not. The reviewing of a document can be done from the first phase of software development i.e. software requirement and analysis phase where the end product is the SRS document. There are many methods for practicing the verification of the software like peer reviews, walkthroughs, inspections, etc that can help us in the prevention of potential faults otherwise, it may lead to the failure of software.

Methods of Verification :

1. Peer Reviews –
The very easiest method and informal way of reviewing the documents or the programs/software for the purpose of finding out the faults during the verification process is the peer-review

method. In this method, we give the document or software programs to others and ask them to review those documents or software programs where we expect their views about the quality of our product and also expect them to find the faults in the program/document. The activities that are involved in this method may include SRS document verification, SDD verification, and program verification. In this method, the reviewers may also prepare a short report on their observations or findings, etc.

Advantages:

- You can expect some good results without spending any significant resources.
- It is very efficient and significant in its nature.

Disadvantages:

- Lead to bad results if the reviewer doesn't have sufficient knowledge.

2. Walk-through –

Walk-throughs are the formal and very systematic type of verification method as compared to peer-review. In a walkthrough, the author of the software document presents the document to other persons which can range from 2 to 7. Participants are not expected to prepare anything. The presenter is responsible for preparing the meeting. The document(s) is/are distributed to all participants. At the time of the meeting of the walk-through, the author introduces the content in order to make them familiar with it and all the participants are free to ask their doubts.

Advantages:

- It may help us to find potential faults.
- It may also be used for sharing documents with others.

Disadvantages:

- The author may hide some critical areas and unnecessarily emphasize some specific areas of his / her interest.

3. Inspections –

Inspections are the most structured and most formal type of verification method and are commonly known as inspections. A team of three to six participants is constituted which is led by an impartial moderator. Every person in the group participates openly, actively, and follows the rules about how such a review is to be conducted. Everyone may get time to express their views, potential faults, and critical areas. After the meeting, a final report is prepared after incorporating necessary suggestions by the moderator.

S.no	Method	Presenter	No. of Members	Pre-requisites	Report	Strength	Weakness
1.	**Peer reviews**	0	1 or 2	No prerequisite	Not Required	Less-Expensive	Output is dependent on the ability of the reviewer
2.	**Walkthrough**	Author	2 to 7 members	The only presenter is required to be prepared	The report is prepared by the presenter	Knowledge sharing	Find few faults and not very expensive
3.	**Inspection**	Author and other members	3 to 6 members	All participants are required to be prepared	The report is prepared by the moderator	Effective but may get faults	Expensive and requires very skilled members

Advantages:

- It can be very effective for finding potential faults or problems in the documents like SRS, SDD, etc.
- The critical inspections may also help in finding faults and improve these documents which can in preventing the propagation of a fault in the software development life cycle process.

Disadvantages:

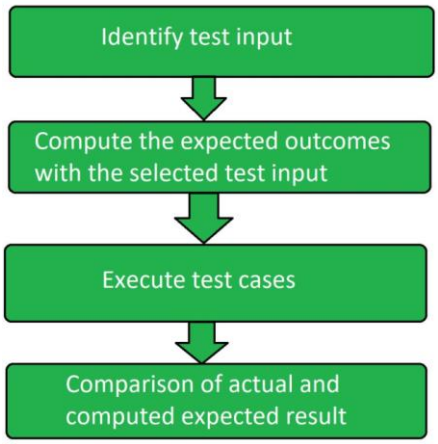
- They take time and require discipline.
- It requires more cost and also needs skilled testers.

Applications of verification methods :
The above three verification methods are very popular and have their own strengths and weaknesses. We can compare these methods on various specific issues as given below:
Hence, Verification is likely more effective than validation but it may find some faults that are somewhat impossible to find during the validation process. But at the same time, it allows us to find faults at the earliest possible phase/time of software development.

Software Testing | Functional Testing
Functional Testing is a type of Software Testing in which the system is tested against the functional requirements and specifications. Functional testing ensures that the requirements or specifications are properly satisfied by the application. This type of testing is particularly concerned with the result of processing. It focuses on simulation of actual system usage but does not develop any system structure assumptions.
It is basically defined as a type of testing which verifies that each function of the software application works in conformance with the requirement and specification. This testing is not concerned about the source code of the application. Each functionality of the software application is tested by providing appropriate test input, expecting the output and comparing the actual output with the expected output. This testing focuses on checking of user interface, APIs, database, security, client or server application and functionality of the Application Under Test.

Functional testing can be manual or automated.

Functional Testing Process:
Functional testing involves the following steps:
1. Identify function that is to be performed.
2. Create input data based on the specifications of function.
3. Determine the output based on the specifications of function.
4. Execute the test case.
5. Compare the actual and expected output.



Major Functional Testing Techniques:

- Unit Testing
- Integration Testing
- Smoke Testing

- User Acceptance Testing
- Interface Testing
- Usability Testing
- System Testing
- Regression Testing

Functional Testing Tools:

1. Selenium
2. QTP
3. JUnit
4. SoapUI
5. Watir

Advantages of Functional Testing:

- It ensures to deliver a bug-free product.
- It ensures to deliver a high-quality product.
- No assumptions about the structure of the system.
- This testing is focused on the specifications as per the customer usage.

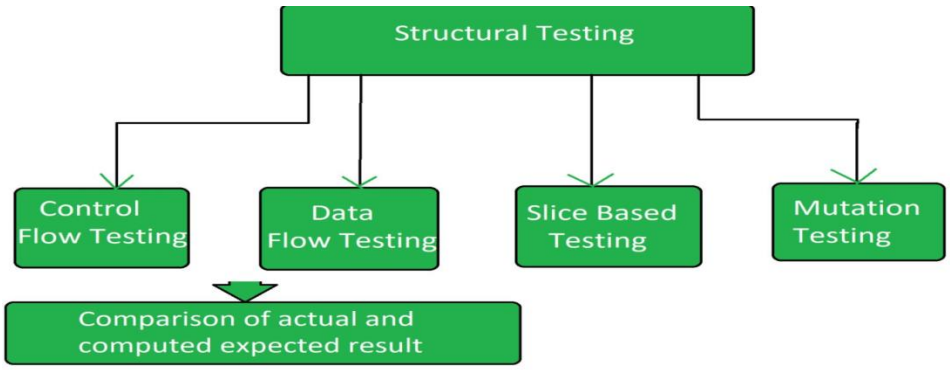
Disadvantages of Functional Testing:

- There are high chances of performing redundant testing.
- Logical errors can be missed out in the product.
- If the requirement is not complete then performing this testing becomes difficult.

Structural Software Testing

Structural testing is a type of software testing which uses the internal design of the software for testing or in other words the software testing which is performed by the team which knows the development phase of the software, is known as structural testing.
Structural testing is basically related to the internal design and implementation of the software i.e. it involves the development team members in the testing team. It basically tests different aspects of the software according to its types. Structural testing is just the opposite of behavioral testing.

Types of Structural Testing:
There are 4 types of Structural Testing:



Control Flow Testing:
Control flow testing is a type of structural testing that uses the programs's control flow as a model. The entire code, design and structure of the software have to be known for this type of testing. Often this type of testing is used by the developers to test their own code and implementation. This method is used to test the logic of the code so that required result can be obtained.

Data Flow Testing:
It uses the control flow graph to explore the unreasonable things that can happen to data.
The detection of data flow anomalies are based on the associations between values and variables. Without being initialized usage of variables. Initialized variables are not used once.

Slice Based Testing:
It was originally proposed by Weiser and Gallagher for the software maintenance. It is useful for software debugging, software maintenance, program understanding and quantification of functional cohesion. It divides the program into different slices and tests that slice which can majorly affect the entire software.

Mutation Testing:
Mutation Testing is a type of Software Testing that is performed to design new software tests and also evaluate the quality of already existing software tests. Mutation testing is related to modification a program in small ways. It focuses to help the tester develop effective tests or locate weaknesses in the test data used for the program.

Advantages of Structural Testing:

- It provides thorough testing of the software.
- It helps in finding out defects at an early stage.
- It helps in elimination of dead code.
- It is not time consuming as it is mostly automated.

Disadvantages of Structural Testing:

- It requires knowledge of the code to perform test.
- It requires training in the tool used for testing.
- Sometimes it is expensive.

Structural Testing Tools:

- JBehave
- Cucumber
- Junit
- Cfix

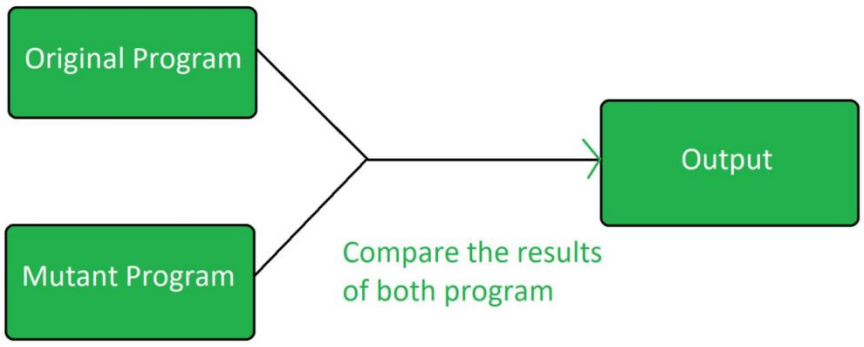
Class Testing **Based on Method Testing**:

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

Mutation Testing is a type of Software Testing that is performed to design new software tests and also evaluate the quality of already existing software tests. Mutation testing is related to modification a program in small ways. It focuses to help the tester develop effective tests or locate weaknesses in the test data used for the program.

History of Mutation Testing:

Richard Lipton proposed the mutation testing in 1971 for the first time. Although high cost reduced the use of mutation testing but now it is widely used for languages such as Java and XML.



Mutation Testing is a White Box Testing.

Mutation testing can be applied to design models, specifications, databases, tests, and XML. It is a structural testing technique, which uses the structure of the code to guide the testing process. It can be described as the process of rewriting the source code in small ways in order to remove the redundancies in the source code.

Objective of Mutation Testing:

The objective of mutation testing is:

- To identify pieces of code that are not tested properly.

- To identify hidden defects that can't be detected using other testing methods.
- To discover new kinds of errors or bugs.
- To calculate the mutation score.
- To study error propagation and state infection in the program.
- To assess the quality of the test cases.

Types of Mutation Testing:

Mutation testing is basically of 3 types:

1. **Value Mutations:**
   In this type of testing the values are changed to detect errors in the program. Basically a small value is changed to a larger value or a larger value is changed to a smaller value. In this testing basically constants are changed.
   Example:
   Initial Code:

   ```
   int mod = 1000000007;
   int a = 12345678;
   int b = 98765432;
   int c = (a + b) % mod;
   ```

   Changed Code:

   ```
   int mod = 1007;
   int a = 12345678;
   int b = 98765432;
   int c = (a + b) % mod;
   ```
2. **Decision Mutations:**
   In decisions mutations are logical or arithmetic operators are changed to detect errors in the program.
   Example:
   Initial Code:

   ```
   if(a < b)
    c = 10;
   else
    c = 20;
   ```

   Changed Code:

   ```
   if(a > b)
    c = 10;
   else
    c = 20;
   ```
3. **Statement Mutations:**
   In statement mutations a statement is deleted or it is replaces by some other statement.
   Example:
   Initial Code:

```
if(a < b)
 c = 10;
else
 c = 20;
```

Changed Code:

```
if(a < b)
 d = 10;
else
 d = 20;
```

Advantages of Mutation Testing:

- It brings a good level of error detection in the program.
- It discovers ambiguities in the source code.

Disadvantages of Mutation Testing:

- It is highly costly and time-consuming.
- It is not able for Black Box Testing.

Levels of Testing in Software Testing

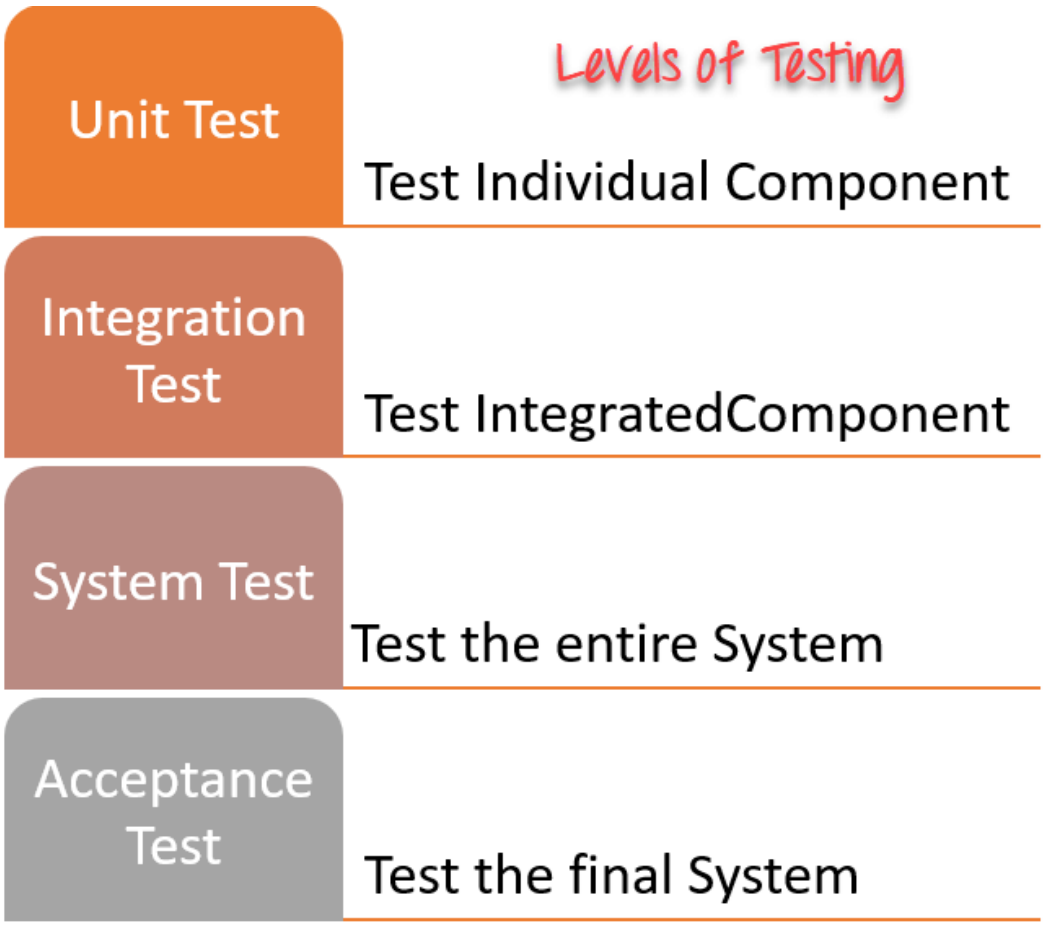
Tests are grouped together based on where they are added in SDLC or the by the level of detailing they contain. In general, there are four levels of testing: unit testing, integration testing, system testing, and acceptance testing. The purpose of Levels of testing is to make software testing systematic and easily identify all possible test cases at a particular level.

There are many different testing levels which help to check behavior and performance for software testing. These testing levels are designed to recognize missing areas and reconciliation between the development lifecycle states. In SDLC models there are characterized phases such as requirement gathering, analysis, design, coding or execution, testing, and deployment. All these phases go through the process of software testing levels.

Levels of Testing

There are mainly four **Levels of Testing** in software testing :

1. **Unit Testing** : checks if software components are fulfilling functionalities or not.
2. **Integration Testing** : checks the data flow from one module to other modules.
3. **System Testing** : evaluates both functional and non-functional needs for the testing.
4. **Acceptance Testing** : checks the requirements of a specification or contract are met as per its delivery.



Each of these testing levels has a specific purpose. These testing level provide value to the software development lifecycle.

1) **Unit testing:**

A Unit is a smallest testable portion of system or application which can be compiled, liked, loaded, and executed. This kind of testing helps to test each module separately.

The aim is to test each part of the software by separating it. It checks that component are fulfilling functionalities or not. This kind of testing is performed by developers.

2) **Integration testing:**

Integration means combining. For Example, In this testing phase, different software modules are combined and tested as a group to make sure that integrated system is ready for system testing.

Integrating testing checks the data flow from one module to other modules. This kind of testing is performed by testers.

3) **System testing:**

System testing is performed on a complete, integrated system. It allows checking system's compliance as per the requirements. It tests the overall interaction of components. It involves load, performance, reliability and security testing.

System testing most often the final test to verify that the system meets the specification. It evaluates both functional and non-functional need for the testing.

4) **Acceptance testing:**

Acceptance testing is a test conducted to find if the requirements of a specification or contract are met as per its delivery. Acceptance testing is basically done by the user or customer. However, other stockholders can be involved in this process.

Other Types of Testing:

- Regression Testing
- Buddy Testing
- Alpha Testing
- Beta Testing

Conclusion:

- A level of software testing is a process where every unit or component of a software/system is tested.
- The primary goal of system testing is to evaluate the system's compliance with the specified needs.
- In Software Engineering, four main levels of testing are Unit Testing, Integration Testing, System Testing and Acceptance Testing.

Difference between Static and Dynamic Testing

Static Testing**:**
Static Testing is a type of a Software Testing method which is performed to check the defects in software without actually executing the code of the software application.
Static testing is performed in early stage of development to avoid errors as it is easier to find sources of failures and it can be fixed easily. The errors that can't not be found using Dynamic Testing, can be easily found by Static Testing.

Dynamic Testing:
Dynamic Testing is a type of Software Testing which is performed to analyze the dynamic behavior of the code. It includes the testing of the software for the input values and output values that are analyzed.
Difference between Static Testing and Dynamic Testing:

Static Testing	Dynamic Testing
It is performed in the early stage of the software development.	It is performed at the later stage of the software development.
In static testing whole code is not executed.	In dynamic testing whole code is executed.
Static testing prevents the defects.	Dynamic testing finds and fixes the defects.
Static testing is performed before code deployment.	Dynamic testing is performed after code deployment.
Static testing is less costly.	Dynamic testing is highly costly.
Static Testing involves checklist for testing process.	Dynamic Testing involves test cases for testing process.
It includes walkthroughs, code review, inspection etc.	It involves functional and nonfunctional testing.
It generally takes shorter time.	It usually takes longer time as it involves running several test cases.
It can discover variety of bugs.	It expose the bugs that are explorable through execution hence discover only limited type of bugs.
Static Testing may complete 100%	While dynamic testing only achieves less than

Static Testing	Dynamic Testing
statement coverage in comparably less time.	50% statement coverage.
Example:	**Example:**
Verification	Validation

Know the Static Versus Dynamic Testing Tools

Since testing is of two types like 1) Static testing 2) Dynamic testing; accordingly the tools used during these testing are also known as

1) Static testing tools

2) Dynamic testing tools

Static testing tools seek to support the static testing process whereas dynamic testing tools support dynamic testing process. It may be noted that static testing is different from dynamic testing.

Few points of differences among static and dynamic testing are as under:

	Static Testing	Dynamic Testing
1	Static testing does not require the actual execution of software.	Dynamic testing involves testing the software by actually executing it.
2	It is more cost effective.	It is less cost effective.
3	It may achieve 100% statement coverage in relatively short time.	It achieves less than 50% statement coverage because it finds bugs only in part of codes those are actually executed.
4	It usually takes shorter time.	It may involve running several test cases, each of which may take longer then compilation.
5	It may uncover variety of bugs.	It uncovers limited type of bugs that are explorable through execution.
6	It can be done before compilation.	It can take place only after executables are ready

Software testing tools are frequently used to ensure consistency, thoroughness and efficiency in testing software products and to fulfil the requirements of planned testing activities. These tools

may facilitate unit (module) testing and subsequent integration testing (e.g., drivers and stubs) as well as commercial software testing tools.

Testing tools can be classified into following two categories:

Static Test Tools: These tools do not involve actual input and output. Rather, they take a symbolic approach to testing, i.e. they do not test the actual execution of the software. These tools include the following: ,

1) Flow analyzers: They ensure consistency in data flow from input to output.

2) Path tests: They find unused code and code with contradictions.

3) Coverage analyzers: It ensures that all logic paths are tested.

4) Interface analyzers: It examines the effects of passing variables and data between modules.

Dynamic Test Tools: These tools test the software system with 'live' data. Dynamic test tools include the following

1) Test driver: It inputs data into a module-under-test (MUT).

2) Test beds: It simultaneously displays source code along with the program under execution.

3) Emulators: The response facilities are used to emulate parts of the system not yet developed.

4) Mutation analyzers: The errors are deliberately 'fed' into the code in order to test fault tolerance of the system.

Software Engineering | Software Maintenance

Software Maintenance is the process of modifying a software product after it has been delivered to the customer. The main purpose of software maintenance is to modify and update software applications after delivery to correct faults and to improve performance.

Need for Maintenance –

Software Maintenance must be performed in order to:

- Correct faults.
- Improve the design.
- Implement enhancements.
- Interface with other systems.
- Accommodate programs so that different hardware, software, system features, and telecommunications facilities can be used.
- Migrate legacy software.
- Retire software.

Challenges in Software Maintenance:

The various challenges in software maintenance are given below:

- The popular age of any software program is taken into consideration up to ten to fifteen years. As software program renovation is open ended and might maintain for decades making it very expensive.
- Older software program's, which had been intended to paintings on sluggish machines with much less reminiscence and garage ability can not maintain themselves tough in opposition to newly coming more advantageous software program on contemporary-day hardware.
- Changes are frequently left undocumented which can also additionally reason greater conflicts in future.
- As era advances, it turns into high priced to preserve vintage software program.
- Often adjustments made can without problems harm the authentic shape of the software program, making it difficult for any next adjustments.

Categories of Software Maintenance –

Maintenance can be divided into the following:

1. **Corrective maintenance:**
   Corrective maintenance of a software product may be essential either to rectify some bugs observed while the system is in use, or to enhance the performance of the system.

2. **Adaptive maintenance:**
   This includes modifications and updations when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware and software.

3. **Perfective maintenance:**
   A software product needs maintenance to support the new features that the users want or to change different types of functionalities of the system according to the customer demands.

4. **Preventive maintenance:**
   This type of maintenance includes modifications and updations to prevent future problems of the software. It goals to attend problems, which are not significant at this moment but may cause serious issues in future.

Reverse Engineering –

Reverse Engineering is processes of extracting knowledge or design information from anything man-made and reproducing it based on extracted information. It is also called back Engineering.

Software Reverse Engineering –

Software Reverse Engineering is the process of recovering the design and the requirements specification of a product from an analysis of it's code. Reverse Engineering is becoming important, since several existing software products, lack proper documentation, are highly unstructured, or their structure has degraded through a series of maintenance efforts.

Why Reverse Engineering?

- Providing proper system documentation.
- Recovery of lost information.
- Assisting with maintenance.
- Facility of software reuse.
- Discovering unexpected flaws or faults.

Used of Software Reverse Engineering –

- Software Reverse Engineering is used in software design, reverse engineering enables the developer or programmer to add new features to the existing software with or without knowing the source code.
- Reverse engineering is also useful in software testing, it helps the testers to study the virus and other malware code .

Software maintenance is widely accepted part of SDLC now a days. It stands for all the modifications and updations done after the delivery of software product. There are number of reasons, why modifications are required, some of them are briefly mentioned below:

- **Market Conditions** - Policies, which changes over the time, such as taxation and newly introduced constraints like, how to maintain bookkeeping, may trigger need for modification.
- **Client Requirements** - Over the time, customer may ask for new features or functions in the software.
- **Host Modifications** - If any of the hardware and/or platform (such as operating system) of the target host changes, software changes are needed to keep adaptability.
- **Organization Changes** - If there is any business level change at client end, such as reduction of organization strength, acquiring another company, organization venturing into new business, need to modify in the original software may arise.

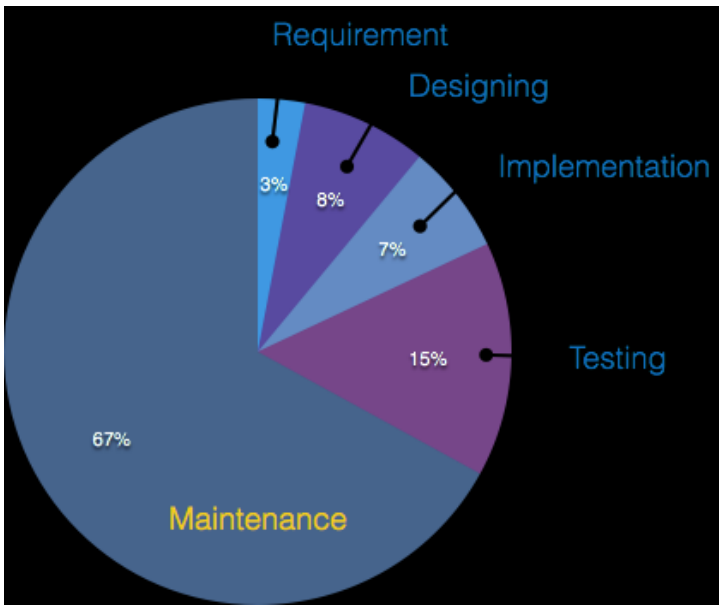
Types of maintenance

In a software lifetime, type of maintenance may vary based on its nature. It may be just a routine maintenance tasks as some bug discovered by some user or it may be a large event in itself based on maintenance size or nature. Following are some types of maintenance based on their characteristics:

- **Corrective Maintenance** - This includes modifications and updations done in order to correct or fix problems, which are either discovered by user or concluded by user error reports.
- **Adaptive Maintenance** - This includes modifications and updations applied to keep the software product up-to date and tuned to the ever changing world of technology and business environment.
- **Perfective Maintenance** - This includes modifications and updates done in order to keep the software usable over long period of time. It includes new features, new user requirements for refining the software and improve its reliability and performance.
- **Preventive Maintenance** - This includes modifications and updations to prevent future problems of the software. It aims to attend problems, which are not significant at this moment but may cause serious issues in future.

Cost of Maintenance

Reports suggest that the cost of maintenance is high. A study on estimating software maintenance found that the cost of maintenance is as high as 67% of the cost of entire software process cycle.



On an average, the cost of software maintenance is more than 50% of all SDLC phases. There are various factors, which trigger maintenance cost go high, such as:

Real-world factors affecting Maintenance Cost

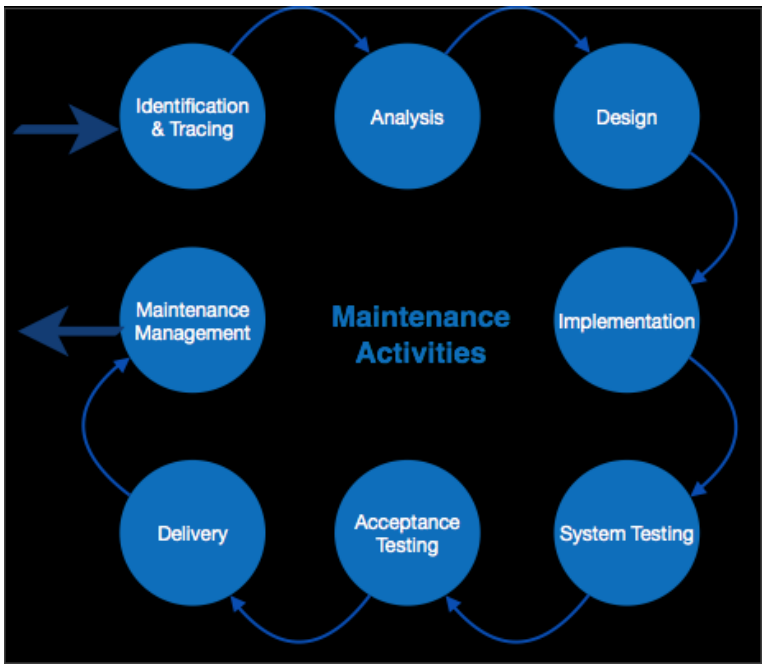
- The standard age of any software is considered up to 10 to 15 years.
- Older softwares, which were meant to work on slow machines with less memory and storage capacity cannot keep themselves challenging against newly coming enhanced softwares on modern hardware.
- As technology advances, it becomes costly to maintain old software.
- Most maintenance engineers are newbie and use trial and error method to rectify problem.
- Often, changes made can easily hurt the original structure of the software, making it hard for any subsequent changes.
- Changes are often left undocumented which may cause more conflicts in future.

Software-end factors affecting Maintenance Cost

- Structure of Software Program
- Programming Language
- Dependence on external environment
- Staff reliability and availability

Maintenance Activities

IEEE provides a framework for sequential maintenance process activities. It can be used in iterative manner and can be extended so that customized items and processes can be included.



These activities go hand-in-hand with each of the following phase:

- **Identification & Tracing** - It involves activities pertaining to identification of requirement of modification or maintenance. It is generated by user or system may itself report via logs or error messages.Here, the maintenance type is classified also.
- **Analysis** - The modification is analyzed for its impact on the system including safety and security implications. If probable impact is severe, alternative solution is looked for. A set of required modifications is then materialized into requirement specifications. The cost of modification/maintenance is analyzed and estimation is concluded.
- **Design** - New modules, which need to be replaced or modified, are designed against requirement specifications set in the previous stage. Test cases are created for validation and verification.
- **Implementation** - The new modules are coded with the help of structured design created in the design step.Every programmer is expected to do unit testing in parallel.
- **System Testing** - Integration testing is done among newly created modules. Integration testing is also carried out between new modules and the system. Finally the system is tested as a whole, following regressive testing procedures.
- **Acceptance Testing** - After testing the system internally, it is tested for acceptance with the help of users. If at this state, user complaints some issues they are addressed or noted to address in next iteration.

- **Delivery** - After acceptance test, the system is deployed all over the organization either by small update package or fresh installation of the system. The final testing takes place at client end after the software is delivered.

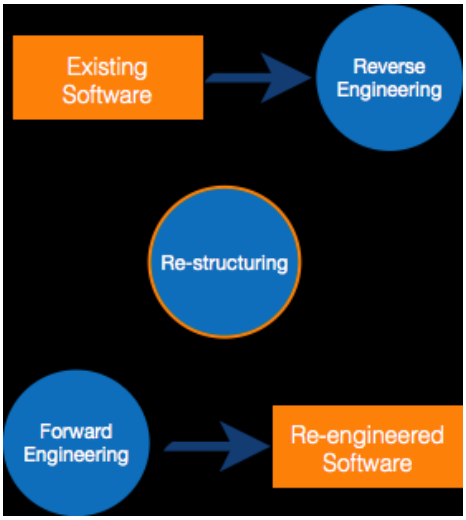
  Training facility is provided if required, in addition to the hard copy of user manual.

- **Maintenance management** - Configuration management is an essential part of system maintenance. It is aided with version control tools to control versions, semi-version or patch management.

Software Re-engineering

When we need to update the software to keep it to the current market, without impacting its functionality, it is called software re-engineering. It is a thorough process where the design of software is changed and programs are re-written.

Legacy software cannot keep tuning with the latest technology available in the market. As the hardware become obsolete, updating of software becomes a headache. Even if software grows old with time, its functionality does not.



For example, initially Unix was developed in assembly language. When language C came into existence, Unix was re-engineered in C, because working in assembly language was difficult.

Other than this, sometimes programmers notice that few parts of software need more maintenance than others and they also need re-engineering.

Re-Engineering Process

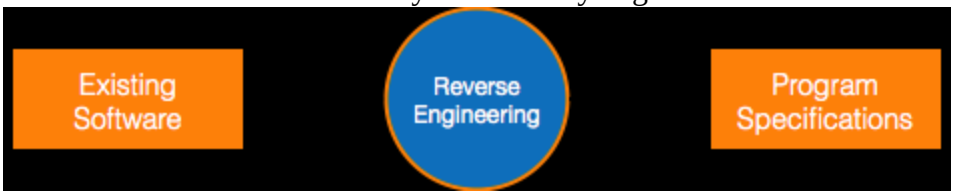
- **Decide** what to re-engineer. Is it whole software or a part of it?

- **Perform** Reverse Engineering, in order to obtain specifications of existing software.
- **Restructure Program** if required. For example, changing function-oriented programs into object-oriented programs.
- **Re-structure data** as required.
- **Apply Forward engineering** concepts in order to get re-engineered software.

There are few important terms used in Software re-engineering

Reverse Engineering

It is a process to achieve system specification by thoroughly analyzing, understanding the existing system. This process can be seen as reverse SDLC model, i.e. we try to get higher abstraction level by analyzing lower abstraction levels.



An existing system is previously implemented design, about which we know nothing. Designers then do reverse engineering by looking at the code and try to get the design. With design in hand, they try to conclude the specifications. Thus, going in reverse from code to system specification.

Program Restructuring

It is a process to re-structure and re-construct the existing software. It is all about re-arranging the source code, either in same programming language or from one programming language to a different one. Restructuring can have either source code-restructuring and data-restructuring or both.

Re-structuring does not impact the functionality of the software but enhance reliability and maintainability. Program components, which cause errors very frequently can be changed, or updated with re-structuring.

The dependability of software on obsolete hardware platform can be removed via re-structuring.

Forward Engineering

Forward engineering is a process of obtaining desired software from the specifications in hand which were brought down by means of reverse engineering. It assumes that there was some software engineering already done in the past.

Forward engineering is same as software engineering process with only one difference – it is carried out always after reverse engineering.

Component reusability

A component is a part of software program code, which executes an independent task in the system. It can be a small module or sub-system itself.

Example

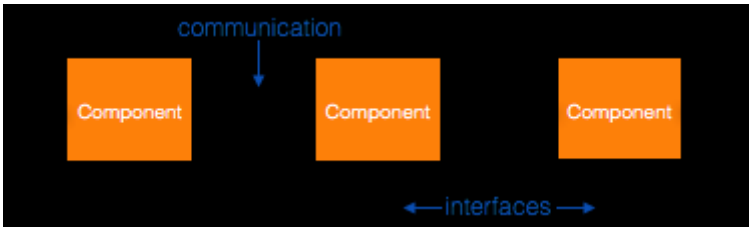
The login procedures used on the web can be considered as components, printing system in software can be seen as a component of the software.

Components have high cohesion of functionality and lower rate of coupling, i.e. they work independently and can perform tasks without depending on other modules.

In OOP, the objects are designed are very specific to their concern and have fewer chances to be used in some other software.

In modular programming, the modules are coded to perform specific tasks which can be used across number of other software programs.

There is a whole new vertical, which is based on re-use of software component, and is known as Component Based Software Engineering (CBSE).



Re-use can be done at various levels

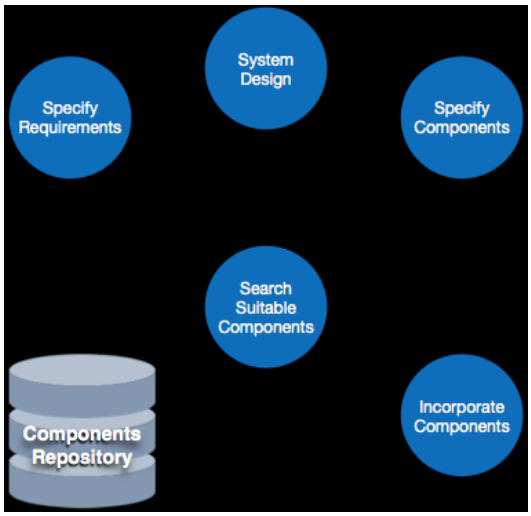
- **Application level** - Where an entire application is used as sub-system of new software.
- **Component level** - Where sub-system of an application is used.
- **Modules level** - Where functional modules are re-used.

  Software components provide interfaces, which can be used to establish communication among different components.

Reuse Process

Two kinds of method can be adopted: either by keeping requirements same and adjusting components or by keeping components same and modifying requirements.

- **Requirement Specification** - The functional and non-functional requirements are specified, which a software product must comply to, with the help of existing system, user input or both.
- **Design** - This is also a standard SDLC process step, where requirements are defined in terms of software parlance. Basic architecture of system as a whole and its sub-systems are created.
- **Specify Components** - By studying the software design, the designers segregate the entire system into smaller components or sub-systems. One complete software design turns into a collection of a huge set of components working together.
- **Search Suitable Components** - The software component repository is referred by designers to search for the matching component, on the basis of functionality and intended software requirements..
- **Incorporate Components** - All matched components are packed together to shape them as complete software.

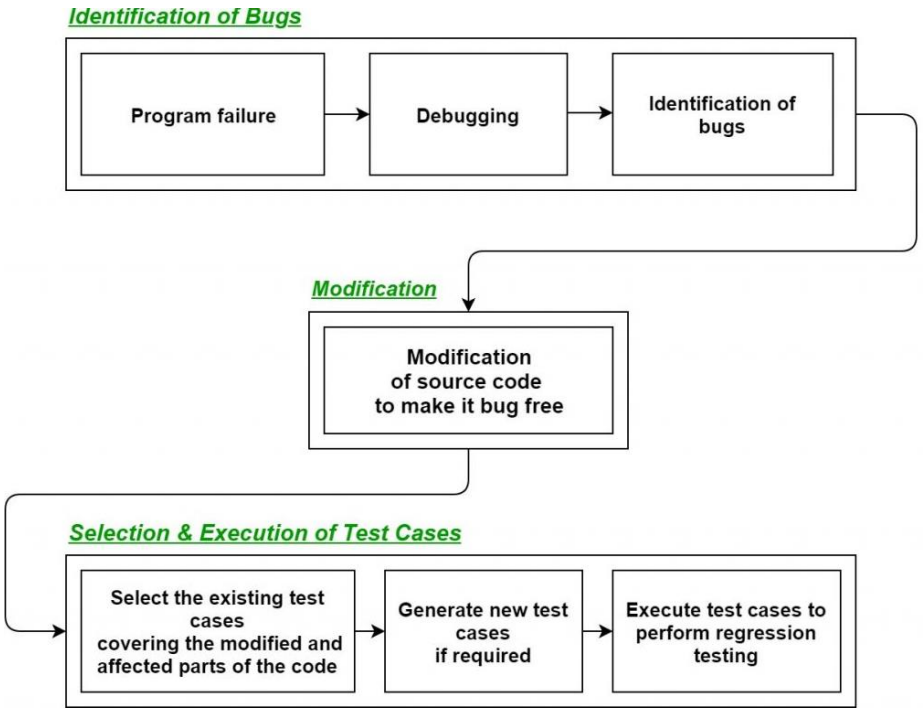


Regression Testing is the process of testing the modified parts of the code and the parts that might get affected due to the modifications to ensure that no new errors have been introduced in the software after the modifications have been made. Regression means return of something and in the software field, it refers to the return of a bug.

When to do regression testing?

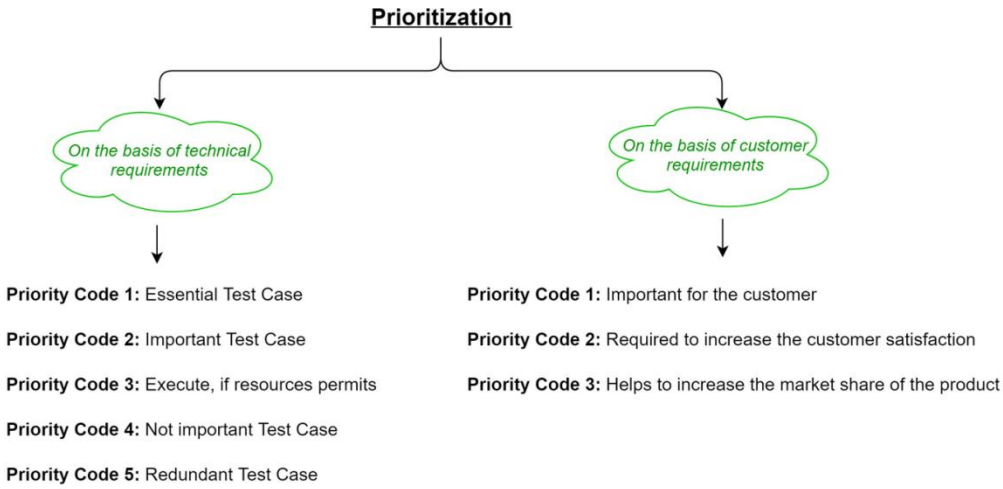
- When a new functionality is added to the system and the code has been modified to absorb and integrate that functionality with the existing code.
- When some defect has been identified in the software and the code is debugged to fix it.
- When the code is modified to optimize its working.

Process of Regression testing:
Firstly, whenever we make some changes to the source code for any reasons like adding new functionality, optimization, etc. then our program when executed fails in the previously designed test suite for obvious reasons. After the failure, the source code is debugged in order to identify the bugs in the program. After identification of the bugs in the source code, appropriate modifications are made. Then appropriate test cases are selected from the already existing test suite which covers all the modified and affected parts of the source code. We can add new test cases if required. In the end regression testing is performed using the selected test cases.



Techniques for the selection of Test cases for Regression Testing:

- **Select all test cases:** In this technique, all the test cases are selected from the already existing test suite. It is the most simple and safest technique but not much efficient.
- **Select test cases randomly:** In this technique, test cases are selected randomly from the existing test-suite but it is only useful if all the test cases are equally good in their fault detection capability which is very rare. Hence, it is not used in most of the cases.
- **Select modification traversing test cases:** In this technique, only those test cases are selected which covers and tests the modified portions of the source code the parts which are affected by these modifications.
- **Select higher priority test cases:** In this technique, priority codes are assigned to each test case of the test suite based upon their bug detection capability, customer requirements, etc. After assigning the priority codes, test cases with highest priorities are selected for the process of regression testing.
  Test case with highest priority has highest rank. For example, test case with priority code 2 is less important than test case with priority code 1.



Tools for regression testing: In regression testing, we generally select the test cases form the existing test suite itself and hence, we need not to compute their expected output and it can be easily automated due to this reason. Automating the process of regression testing will be very much effective and time saving.
Most commonly used tools for regression testing are:

- Selenium
- WATIR (Web Application Testing In Ruby)
- QTP (Quick Test Professional)
- RFT (Rational Functional Tester)
- Winrunner
- Silktest

Advantages of Regression Testing:

- It ensures that no new bugs has been introduced after adding new functionalities to the system.
- As most of the test cases used in Regression Testing are selected from the existing test suite and we already know their expected outputs. Hence, it can be easily automated by the automated tools.
- It helps to maintain the quality of the source code.

Disadvantages of Regression Testing:

- It can be time and resource consuming if automated tools are not used.
- It is required even after very small changes in the code.

UNIT -5: NEED OBJECT-ORIENTED SOFTWARE ESTIMATION

The Object-Oriented Project Size Estimation (Oopsize) technique uses the initial estimates of B1 and B2 **to predict how much time is required to design, code and test an object**. The objects can be described in a Rational Rose class model, for instance.

The calculation of **test estimation techniques** is based on:

- Past Data/Past experience
- Available documents/Knowledge
- Assumptions
- Calculated risks

Software Estimation Techniques

There are different Software **Testing Estimation** Techniques which can be used for estimating a task.

1) Delphi Technique
2) Work Breakdown Structure (WBS)
3) Three Point Estimation
4) Functional Point Method

Disadvantages of **Software Estimation Techniques**:

- Due to hidden factors can be over or under estimated
- Not really accurate
- It is based on thinking
- Involved Risk
- May give false result
- Bare to losing
- Sometimes cannot trust in estimate

Object Oriented Metrices in Software Engineering

These are used to determine success or failure of a person also to quantify the improvements in the software throughout its process. These metrics can be used to reinforce good OO programming technique which lead to more reliable code. Object-oriented software engineering metrics are units of measurement that are used to characterize:

- object-oriented software engineering products, e.g., designs source code, and the test cases.
- object-oriented software engineering processes, e.g., designing and coding.
- object-oriented software engineering people, e.g., productivity of an individual designer.

SOFTWARE MEASUREMENT:

A measurement is a manifestation of the size, quantity, amount or dimension of a particular attribute of a product or process. Software measurement is a titrate impute of a characteristic of a software product or the software process. It is an authority within software engineering. The software measurement process is defined and governed by ISO Standard.

Need of Software Measurement:

Software is measured to:

1. Create the quality of the current product or process.
2. Anticipate future qualities of the product or process.
3. Enhance the quality of a product or process.
4. Regulate the state of the project in relation to budget and schedule.

Classification of Software Measurement:

There are 2 types of software measurement:

1. **Direct Measurement:**
   In direct measurement the product, process or thing is measured directly using standard scale.
2. **Indirect Measurement:**
   In indirect measurement the quantity or quality to be measured is measured using related parameter i.e. by use of reference.

Metrics:

A metric is a measurement of the level that any impute belongs to a system product or process.
There are 4 functions related to software metrics:

1. Planning
2. Organizing
3. Controlling
4. Improving

Characteristics of software Metrics:

1. **Quantitative:**
   Metrics must possess quantitative nature.It means metrics can be expressed in values.
2. **Understandable:**
   Metric computation should be easily understood ,the method of computing metric should be clearly defined.
3. **Applicability:**
   Metrics should be applicable in the initial phases of development of the software.
4. **Repeatable:**
   The metric values should be same when measured repeatedly and consistent in nature.
5. **Economical:**
   Computation of metrics should be economical.
6. **Language Independent:**
   Metrics should not depend on any programming language.

Classification of Software Metrics:

There are 3 types of software metrics:

1. **Product Metrics:**
   Product metrics are used to evaluate the state of the product, tracing risks and undercovering prospective problem areas. The ability of team to control quality is evaluated.
2. **Process Metrics:**
   Process metrics pay particular attention on enhancing the long term process of the team or organization.
3. **Project Metrics:**
   The project matrix describes the project characteristic and execution process.
   - Number of software developer

- Staffing pattern over the life cycle of software
- Cost and schedule
- Productivity

SOFTWARE ENGINEERING | FUNCTIONAL POINT (FP) ANALYSIS

Function Point Analysis was initially developed by Allan J. Albercht in 1979 at IBM and it has been further modified by the International Function Point Users Group (IFPUG).

The initial **Definition** is given **by Allan J. Albrecht:**
FPA gives a dimensionless number defined in function points which we have found to be an effective relative measure of function value delivered to our customer.

FPA provides a standardized method to functionally size the software work product. This work product is the output of software new development and improvement projects for subsequent releases. It is the software that is relocated to the production application at project implementation. It measures functionality from the user's point of view i.e. on the basis of what the user requests and receives in return.

Function Point Analysis (FPA) is a method or set of rules of Functional Size Measurement. It assesses the functionality delivered to its users, based on the user's external view of the functional requirements. It measures the logical view of an application, not the physically implemented view or the internal technical view.

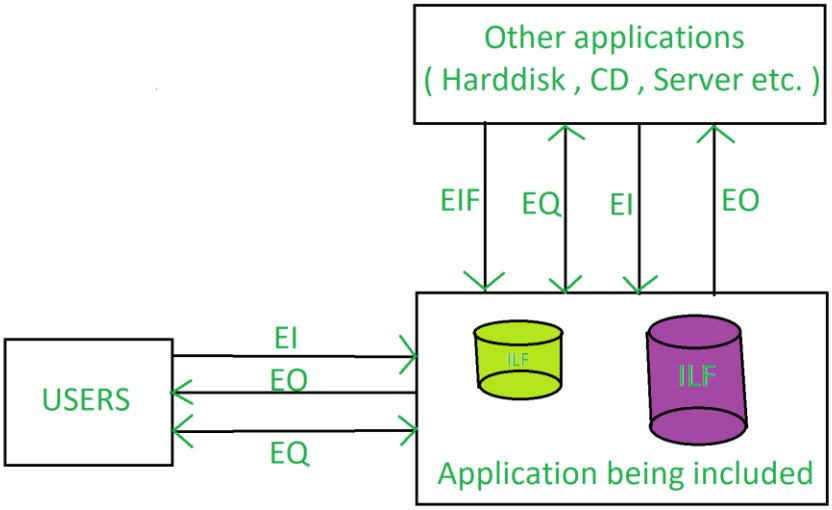
The Function Point Analysis technique is used to analyze the functionality delivered by software and Unadjusted Function Point (UFP) is the unit of measurement.

Objectives of FPA:
- The objective of FPA is to measure the functionality that the user requests and receives.
- The objective of FPA is to measure software development and maintenance independently of the technology used for implementation.
- It should be simple enough to minimize the overhead of the measurement process.
- It should be a consistent measure among various projects and organizations.

Types of FPA:

- **Transactional Functional Type –**
    - **External Input (EI):** EI processes data or control information that comes from outside the application's boundary. The EI is an elementary process.
    - **External Output (EO):** EO is an elementary process that generates data or control information sent outside the application's boundary.

- **External Inquiries (EQ):** EQ is an elementary process made up of an input-output combination that results in data retrieval.



Data
Functional Type –

- **Internal Logical File (ILF):** A user identifiable group of logically related data or control information maintained within the boundary of the application.

- **External Interface File (EIF):** A group of users recognizable logically related data allusion to the software but maintained within the boundary of another software.

Benefits of FPA:

- FPA is a tool to determine the size of a purchased application package by counting all the functions included in the package.
- It is a tool to help users discover the benefit of an application package to their organization by counting functions that specifically match their requirements.
- It is a tool to measure the units of a software product to support quality and productivity analysis.
- It s a vehicle to estimate the cost and resources required for software development and maintenance.
- It is a normalization factor for software comparison.

The drawback of FPA:

- It requires a subjective evaluation and involves many judgements.

- Many cost and effort models are based on LOC, so it is necessary to change the function points.
- Compared to LOC, there are less research data on function points.
- Run after creating the design spec.
- With subjective judgement, the accuracy rate of the assessment is low.
- Due to the long learning curve, it is not easy to gain proficiency.
- This is a very time-consuming method.

A **Use-Case** is a series of related interactions between a user and a system that enables the user to achieve a goal.

Use-Cases are a way to capture functional requirements of a system. The user of the system is referred to as an 'Actor'. Use-Cases are fundamentally in text form.

USE-CASE POINTS – DEFINITION

Use-Case Points (UCP) is a software estimation technique used to measure the software size with use cases. The concept of UCP is similar to FPs.

The number of UCPs in a project is based on the following −

- The number and complexity of the use cases in the system.
- The number and complexity of the actors on the system.
    - Various non-functional requirements (such as portability, performance, maintainability) that are not written as use cases.
    - The environment in which the project will be developed (such as the language, the team's motivation, etc.)

Estimation with UCPs requires all use cases to be written with a goal and at approximately the same level, giving the same amount of detail. Hence, before estimation, the project team should ensure they have written their use cases with defined goals and at detailed level. Use case is normally completed within a single session and after the goal is achieved, the user may go on to some other activity.

History of Use-Case Points

The Use-Case Point estimation method was introduced by Gustav Karner in 1993. The work was later licensed by Rational Software that merged into IBM.

Use-Case Points Counting Process

The Use-Case Points counting process has the following steps −

- Calculate unadjusted UCPs
- Adjust for technical complexity
- Adjust for environmental complexity
- Calculate adjusted UCPs

Step 1: Calculate Unadjusted Use-Case Points.

You calculate Unadjusted Use-Case Points first, by the following steps −

- Determine Unadjusted Use-Case Weight
- Determine Unadjusted Actor Weight
- Calculate Unadjusted Use-Case Points

Step 1.1 − Determine Unadjusted Use-Case Weight.

Step 1.1.1 − Find the number of transactions in each Use-Case.

If the Use-Cases are written with User Goal Levels, a transaction is equivalent to a step in the Use-Case. Find the number of transactions by counting the steps in the Use-Case.

Step 1.1.2 − Classify each Use-Case as Simple, Average or Complex based on the number of transactions in the Use-Case. Also, assign Use-Case Weight as shown in the following table −

Use-Case Complexity	Number of Transactions	Use-Case Weight
Simple	≤3	5
Average	4 to 7	10
Complex	>7	15

Step 1.1.3 − Repeat for each Use-Case and get all the Use-Case Weights. Unadjusted Use-Case Weight (UUCW) is the sum of all the Use-Case Weights.

Step 1.1.4 − Find Unadjusted Use-Case Weight (UUCW) using the following table −

Use-Case Complexity	Use-Case Weight	Number of Use-Cases	Product
Simple	5	NSUC	5 × NSUC
Average	10	NAUC	10 × NAUC

Complex	15	NCUC	15 × NCUC
Unadjusted Use-Case Weight (UUCW)			5 × NSUC + 10 × NAUC + 15 × NCUC

Where,

NSUC is the no. of Simple Use-Cases.

NAUC is the no. of Average Use-Cases.

NCUC is the no. of Complex Use-Cases.

Step 1.2 − Determine Unadjusted Actor Weight.

An Actor in a Use-Case might be a person, another program, etc. Some actors, such as a system with defined API, have very simple needs and increase the complexity of a Use-Case only slightly.

Some actors, such as a system interacting through a protocol have more needs and increase the complexity of a Use-Case to a certain extent.

Other Actors, such as a user interacting through GUI have a significant impact on the complexity of a Use-Case. Based on these differences, you can classify actors as Simple, Average and Complex.

Step 1.2.1 − Classify Actors as Simple, Average and Complex and assign Actor Weights as shown in the following table −

Actor Complexity	Example	Actor Weight
Simple	A System with defined API	1
Average	A System interacting through a Protocol	2
Complex	A User interacting through GUI	3

Step 1.2.2 − Repeat for each Actor and get all the Actor Weights. Unadjusted Actor Weight (UAW) is the sum of all the Actor Weights.

Step 1.2.3 − Find Unadjusted Actor Weight (UAW) using the following table −

Actor Complexity	Actor Weight	Number of Actors	Product
Simple	1	NSA	1 × NSA
Average	2	NAA	2 × NAA
Complex	3	NCA	3 × NCA
Unadjusted Actor Weight (UAW)			1 × NSA + 2 × NAA + 3 × NCA

Where,

NSA is the no. of Simple Actors.

NAA is the no. of Average Actors.

NCA is the no. of Complex Actors.

Step 1.3 − Calculate Unadjusted Use-Case Points.

The Unadjusted Use-Case Weight (UUCW) and the Unadjusted Actor Weight (UAW) together give the unadjusted size of the system, referred to as Unadjusted Use-Case Points.

Unadjusted Use-Case Points (UUCP) = UUCW + UAW

The next steps are to adjust the Unadjusted Use-Case Points (UUCP) for Technical Complexity and Environmental Complexity.

Step 2: Adjust For Technical Complexity

Step 2.1 − Consider the 13 Factors that contribute to the impact of the Technical Complexity of a project on Use-Case Points and their corresponding Weights as given in the following table −

Factor	Description	Weight
T1	Distributed System	2.0
T2	Response time or throughput performance objectives	1.0

T3	End user efficiency	1.0
T4	Complex internal processing	1.0
T5	Code must be reusable	1.0
T6	Easy to install	.5
T7	Easy to use	.5
T8	Portable	2.0
T9	Easy to change	1.0
T10	Concurrent	1.0
T11	Includes special security objectives	1.0
T12	Provides direct access for third parties	1.0
T13	Special user training facilities are required	1.0

Many of these factors represent the project's nonfunctional requirements.

Step 2.2 − For each of the 13 Factors, assess the project and rate from 0 (irrelevant) to 5 (very important).

Step 2.3 − Calculate the Impact of the Factor from Impact Weight of the Factor and the Rated Value for the project as

Impact of the Factor = Impact Weight × Rated Value

Step (2.4) − Calculate the sum of Impact of all the Factors. This gives the Total Technical Factor (TFactor) as given in table below −

Factor	Description	Weight (W)	Rated Value (0 to 5) (RV)	Impact (I = W × RV)
T1	Distributed System	2.0		
T2	Response time or throughput performance objectives	1.0		
T3	End user efficiency	1.0		
T4	Complex internal processing	1.0		
T5	Code must be reusable	1.0		
T6	Easy to install	.5		
T7	Easy to use	.5		
T8	Portable	2.0		
T9	Easy to change	1.0		
T10	Concurrent	1.0		
T11	Includes special security objectives	1.0		
T12	Provides direct access for third parties	1.0		

T13	Special user training facilities are required	1.0		
Total Technical Factor (TFactor)				

Step 2.5 − Calculate the Technical Complexity Factor (TCF) as −

TCF = 0.6 + (0.01 × TFactor)

Step 3: Adjust For Environmental Complexity

Step 3.1 − Consider the 8 Environmental Factors that could affect the project execution and their corresponding Weights as given in the following table −

Factor	Description	Weight
F1	Familiar with the project model that is used	1.5
F2	Application experience	.5
F3	Object-oriented experience	1.0
F4	Lead analyst capability	.5
F5	Motivation	1.0
F6	Stable requirements	2.0
F7	Part-time staff	-1.0
F8	Difficult programming language	-1.0

Step 3.2 − For each of the 8 Factors, assess the project and rate from 0 (irrelevant) to 5 (very important).

Step 3.3 − Calculate the Impact of the Factor from Impact Weight of the Factor and the Rated Value for the project as

Impact of the Factor = Impact Weight × Rated Value

Step 3.4 − Calculate the sum of Impact of all the Factors. This gives the Total Environment Factor (EFactor) as given in the following table −

Factor	Description	Weight (W)	Rated Value (0 to 5) (RV)	Impact (I = W × RV)
F1	Familiar with the project model that is used	1.5		
F2	Application experience	.5		
F3	Object-oriented experience	1.0		
F4	Lead analyst capability	.5		
F5	Motivation	1.0		
F6	Stable requirements	2.0		
F7	Part-time staff	-1.0		
F8	Difficult programming language	-1.0		
Total Environment Factor (EFactor)				

Step 3.5 − Calculate the Environmental Factor (EF) as −

1.4 + (-0.03 × EFactor)

Step 4: Calculate Adjusted Use-Case Points (UCP)

Calculate Adjusted Use-Case Points (UCP) as −

UCP = UUCP × TCF × EF

Advantages and Disadvantages of Use-Case Points

Advantages of Use-Case Points

- UCPs are based on use cases and can be measured very early in the project life cycle.
- UCP (size estimate) will be independent of the size, skill, and experience of the team that implements the project.
- UCP based estimates are found to be close to actuals when estimation is performed by experienced people.
- UCP is easy to use and does not call for additional analysis.
- Use cases are being used vastly as a method of choice to describe requirements. In such cases, UCP is the best suitable estimation technique.

Disadvantages of Use-Case Points

- UCP can be used only when requirements are written in the form of use cases.
- Dependent on goal-oriented, well-written use cases. If the use cases are not well or uniformly structured, the resulting UCP may not be accurate.
- Technical and environmental factors have a high impact on UCP. Care needs to be taken while assigning values to the technical and environmental factors.
- UCP is useful for initial estimate of overall project size but they are much less useful in driving the iteration-to-iteration work of a team.

RISK MANAGEMENT

"Tomorrow problems are today's risk." Hence, a clear definition of a "risk" is a problem that could cause some loss or threaten the progress of the project, but which has not happened yet.

These potential issues might harm cost, schedule or technical success of the project and the quality of our software device, or project team morale.

Risk Management is the system of identifying addressing and eliminating these problems before they can damage the project.

We need to differentiate risks, as potential issues, from the current problems of the project.

Different methods are required to address these two kinds of issues.

For example, staff storage, because we have not been able to select people with the right technical skills is a current problem, but the threat of our technical persons being hired away by the competition is a risk.

Risk Management

A software project can be concerned with a large variety of risks. In order to be adept to systematically identify the significant risks which might affect a software project, it is essential to classify risks into different classes. The project manager can then check which risks from each class are relevant to the project.

There are three main classifications of risks which can affect a software project:

1. Project risks
2. Technical risks
3. Business risks

1. Project risks: Project risks concern differ forms of budgetary, schedule, personnel, resource, and customer-related problems. A vital project risk is schedule slippage. Since the software is intangible, it is very tough to monitor and control a software project. It is very tough to control something which cannot be identified. For any manufacturing program, such as the manufacturing of cars, the plan executive can recognize the product taking shape.

2. Technical risks: Technical risks concern potential method, implementation, interfacing, testing, and maintenance issue. It also consists of an ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence. Most technical risks appear due to the development team's insufficient knowledge about the project.

3. Business risks: This type of risks contain risks of building an excellent product that no one need, losing budgetary or personnel commitments, etc.

Other risk categories

1. **1. Known risks:** Those risks that can be uncovered after careful assessment of the project program, the business and technical environment in which the plan is being developed, and more reliable data sources (e.g., unrealistic delivery date)
2. **2. Predictable risks:** Those risks that are hypothesized from previous project experience (e.g., past turnover)

3. **3. Unpredictable risks:** Those risks that can and do occur, but are extremely tough to identify in advance.

Principle of Risk Management

1. **Global Perspective:** In this, we review the bigger system description, design, and implementation. We look at the chance and the impact the risk is going to have.
2. **Take a forward-looking view:** Consider the threat which may appear in the future and create future plans for directing the next events.
3. **Open Communication:** This is to allow the free flow of communications between the client and the team members so that they have certainty about the risks.
4. **Integrated management:** In this method risk management is made an integral part of project management.
5. **Continuous process:** In this phase, the risks are tracked continuously throughout the risk management paradigm.

Risk Management Activities

Risk management consists of three main activities, as shown in fig:

Risk Assessment

The objective of risk assessment is to division the risks in the condition of their loss, causing potential. For risk assessment, first, every risk should be rated in two methods:

- The possibility of a risk coming true (denoted as r).
- The consequence of the issues relates to that risk (denoted as s).

Based on these two methods, the priority of each risk can be estimated:

$$p = r * s$$

Where p is the priority with which the risk must be controlled, r is the probability of the risk becoming true, and s is the severity of loss caused due to the risk becoming true. If all identified risks are set up, then the most likely and damaging risks can be controlled first, and more comprehensive risk abatement methods can be designed for these risks.

1. Risk Identification: The project organizer needs to anticipate the risk in the project as early as possible so that the impact of risk can be reduced by making effective risk management planning.

A project can be of use by a large variety of risk. To identify the significant risk, this might affect a project. It is necessary to categories into the different risk of classes.

There are different types of risks which can affect a software project:

1. **Technology risks:** Risks that assume from the software or hardware technologies that are used to develop the system.
2. **People risks:** Risks that are connected with the person in the development team.
3. **Organizational risks:** Risks that assume from the organizational environment where the software is being developed.
4. **Tools risks:** Risks that assume from the software tools and other support software used to create the system.
5. **Requirement risks:** Risks that assume from the changes to the customer requirement and the process of managing the requirements change.

6. **Estimation risks:** Risks that assume from the management estimates of the resources required to build the system

2. Risk Analysis: During the risk analysis process, you have to consider every identified risk and make a perception of the probability and seriousness of that risk.

There is no simple way to do this. You have to rely on your perception and experience of previous projects and the problems that arise in them.

It is not possible to make an exact, the numerical estimate of the probability and seriousness of each risk. Instead, you should authorize the risk to one of several bands:

1. The probability of the risk might be determined as very low (0-10%), low (10-25%), moderate (25-50%), high (50-75%) or very high (+75%).
2. The effect of the risk might be determined as catastrophic (threaten the survival of the plan), serious (would cause significant delays), tolerable (delays are within allowed contingency), or insignificant.

Risk Control

It is the process of managing risks to achieve desired outcomes. After all, the identified risks of a plan are determined; the project must be made to include the most harmful and the most likely risks. Different risks need different containment methods. In fact, most risks need ingenuity on the part of the project manager in tackling the risk.

There are three main methods to plan for risk management:

1. **Avoid the risk:** This may take several ways such as discussing with the client to change the requirements to decrease the scope of the work, giving incentives to the engineers to avoid the risk of human resources turnover, etc.
2. **Transfer the risk:** This method involves getting the risky element developed by a third party, buying insurance cover, etc.
3. **Risk reduction:** This means planning method to include the loss due to risk. For instance, if there is a risk that some key personnel might leave, new recruitment can be planned.

Risk Leverage: To choose between the various methods of handling risk, the project plan must consider the amount of controlling the risk and the corresponding reduction of risk. For this, the risk leverage of the various risks can be estimated.

Risk leverage is the variation in risk exposure divided by the amount of reducing the risk.

Risk leverage = (risk exposure before reduction - risk exposure after reduction) / (cost of reduction)

1. Risk planning: The risk planning method considers each of the key risks that have been identified and develop ways to maintain these risks.

For each of the risks, you have to think of the behavior that you may take to minimize the disruption to the plan if the issue identified in the risk occurs.

You also should think about data that you might need to collect while monitoring the plan so that issues can be anticipated.

Again, there is no easy process that can be followed for contingency planning. It rely on the judgment and experience of the project manager.

3. **Risk Monitoring:** Risk monitoring is the method king that your assumption about the product, process, and business risks has not changed.

MEASURING SOFTWARE QUALITY USING QUALITY METRICS

In Software Engineering, Software Measurement is done based on some Software Metrics where these software metrics are referred to as the measure of various characteristics of a Software.

In Software engineering Software Quality Assurance (SAQ) assures the quality of the software. Set of activities in SAQ are continuously applied throughout the software process. Software Quality is measured based on some software quality metrics.

There is a number of metrics available based on which software quality is measured. But among them, there are few most useful metrics which are most essential in software quality measurement. They are –

1. Code Quality
2. Reliability
3. Performance
4. Usability
5. Correctness
6. Maintainability
7. Integrity
8. Security

Now let's understand each quality metric **in detail –**

1. **Code Quality –** Code quality metrics measure the quality of code used for the software project development. Maintaining the software code quality by writing Bug-free and semantically correct code is very important for good software project development. In code

quality both Quantitative metrics like the number of lines, complexity, functions, rate of bugs generation, etc, and Qualitative metrics like readability, code clarity, efficiency, maintainability, etc are measured.

2. **Reliability –** Reliability metrics express the reliability of software in different conditions. The software is able to provide exact service at the right time or not is checked. Reliability can be checked using Mean Time Between Failure (MTBF) and Mean Time To Repair (MTTR).

3.Performance – Performance metrics are used to measure the performance of the software. Each software has been developed for some specific purposes. Performance metrics measure the performance of the software by determining whether the software is fulfilling the user requirements or not, by analyzing how much time and resource it is utilizing for providing the service.

4.Usability – Usability metrics check whether the program is user-friendly or not. Each software is used by the end-user. So it is important to measure that the end-user is happy or not by using this software.

5. Correctness – Correctness is one of the important software quality metrics as this checks whether the system or software is working correctly without any error by satisfying the user. Correctness gives the degree of service each function provides as per developed.

6. Maintainability – Each software product requires maintenance and up-gradation. Maintenance is an expensive and time-consuming process. So if the software product provides easy maintainability then we can say software quality is up to mark. Maintainability metrics include time requires to adapt to new features/functionality, Mean Time to Change (MTTC), performance in changing environments, etc.

7. Integrity – Software integrity is important in terms of how much it is easy to integrate with other required software's which increases software functionality and what is the control on integration from unauthorized software's which increases the chances of cyberattacks.

8. Security – Security metrics measure how much secure the software is? In the age of cyber terrorism, security is the most essential part of every software. Security assures that there are no unauthorized changes, no fear of cyber attacks, etc when the software product is in use by the end-user.