**Branch :- Computer Sci. & Engg. Class :- III Year**

**Subject :- Software Engineering Lab Sem :- VI**

**Teacher Manual**

**PRACTICAL NO 1**

**Aim:** Study of various phases of SDLC.

**Theory:**

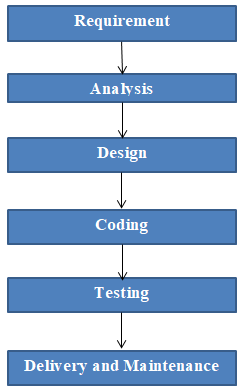
“SDLC is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software”

**Need of Software Development Life Cycle**

“SDLC ensure success in process of software development.”

**Phases of Software Development Life Cycle**

* **Initial**
* **Analysis**
* **Design**
* **Coding**
* **Testing**
* **Delivery & Maintenance**



**Figure1: phases of SDLC**

**Initial**

“Business requirements are gathered in this phase. “

This phase is the main focus of the project managers and stake holders. Meetings with managers, stake holders and users are held in order to determine the requirements like;

- Who is going to use the system?

- How will they use the system?

- What data should be input into the system?

- What data should be output by the system?

**Roles Involved:** Business Analyst (BA), System Architects

## Outcome: System Requirement Specification (SRS

**Analysis**

“After requirement gathering these requirements are analyzed for their validity and the possibility of developing the requirements in the system.”

Requirement analysis is the most important and fundamental stage in SDLC. It is performed by both development team and testing team.

**Roles Involved:** Developer & Quality Analysis team, Architects, Project Managers

**Outcome:** Final SRS approved by customer, Technology selection for both Developer & QA

**Design**

“During this part of the design phase, the consultants/architects break down the system into pieces that can be programmed.”

System Design helps in specifying hardware and system requirements and also helps in defining overall

system architecture. The system design specifications serve as input for the next phase of the model.

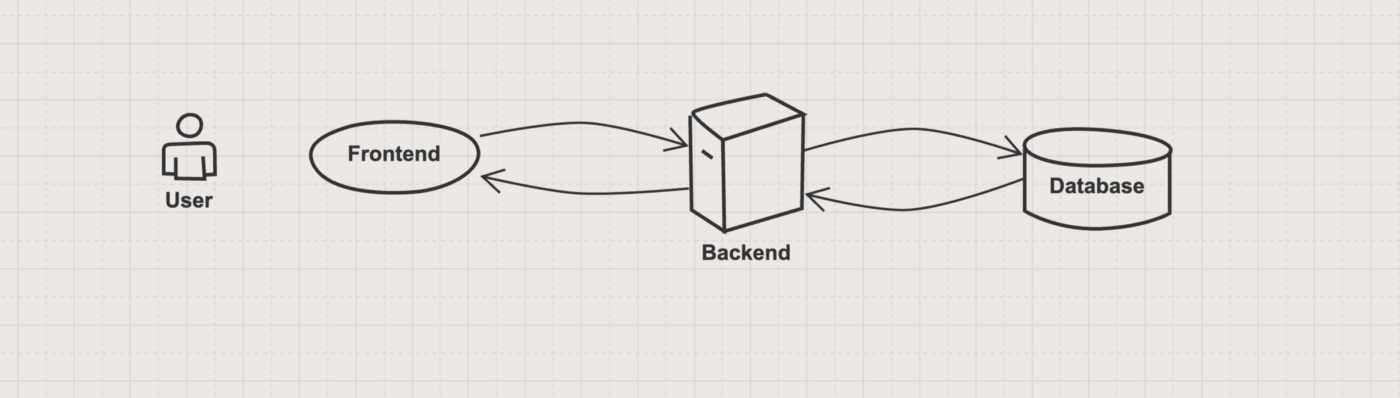
**Roles Involved:** Architects & Team

**Outcome**: Technical Design Document (TDD)

**Coding**

“The actual development starts and the product is built in coding phase. “

The work is divided in modules/units and actual coding is started in this coding phase and it is the main focus for developer. Coding is one of the longest phase of SDLC.



**Figure 2: Connection between Frontend, Backend and Databases**

**Roles involved:** Developers and Architects

**Outcome:** Programs or Application or Module

**Testing**

“In Testing phase testers execute the test cases against the application, report the defects and retested

the fixed defects. “

During this phase unit testing, integration testing, system testing, acceptance testing are done.

**Roles Involved:** Testers, Developers

**Outcome:** Defects, Test Summary Report, Test Plan, Test Case document

**Delivery & Maintenance**

“After successful testing the product is delivered / deployed to the customer.”

During the Delivery phase, customer will perform user acceptance testing (UAT) in a real time environment.

Once when the customers starts using the developed system then the actual problems comes up and needs to be solved from time to time. This process where the care is taken for the developed product is known as maintenance.

**Roles Involved:** Testers, Developers, Customer, Business team, Architects, Project Manager, and Delivery Manager

**Outcome:** Quality Product, Enhancements & Production Issues (Maintenance)

**Conclusion:** Thus, we have studied the SDLC and understand the Roles, Responsibility and outcome of SDLC phases

**Teacher Manual**

**PRACTICAL NO 2**

**Aim:** [Identifying the Requirements from Problem Statements](http://vlabs.iitkgp.ernet.in/se/1/)

**Introduction**

Requirements identification is the first step of any software development project. Until the requirements of a client have been clearly identified, and verified, no other task (design, coding, testing) could begin. Usually business analysts having domain knowledge on the subject matter discuss with clients and decide what features are to be implemented.

In this experiment we will learn how to identify functional and non-functional requirements from a given problem statement. Functional and non-functional requirements are the primary components of a Software Requirements Specification.

## Objectives

**After completing this experiment you will be able to:**

* Identify ambiguities, inconsistencies and incompleteness from a requirements specification
* Identify and state functional requirements
* Identify and state non-functional requirements

## Requirements:

Requirements specify how the target system should behave. It specifies what to do, but not how to do. Requirements engineering refers to the process of understanding what a customer expects from the system to be developed, and to document them in a standard and easily readable and understandable format. This documentation will serve as reference for the subsequent design, implementation and verification of the system.

It is necessary and important that before we start planning, design and implementation of the software system for our client, we are clear about it's requirements. If we don't have a clear vision of what is to be developed and what all features are expected, there would be serious problems, and customer dissatisfaction as well.

## Characteristics of Requirements

Requirements gathered for any new system to be developed should exhibit the following three properties:

* **Unambiguity:** There should not be any ambiguity what a system to be developed should do. For example, consider you are developing a web application for your client. The client requires that enough number of people should be able to access the application simultaneously. What's the "enough number of people"? That could mean 10 to you, but, perhaps, 100 to the client. There's an ambiguity.
* **Consistency:** To illustrate this, consider the automation of a nuclear plant. Suppose one of the clients say that it the radiation level inside the plant exceeds R1, all reactors should be shut down. However, another person from the client side suggests that the threshold radiation level should be R2. Thus, there is an inconsistency between the two end users regarding what they consider as threshold level of radiation.
* **Completeness:** A particular requirement for a system should specify what the system should do and also what it should not. For example, consider a software to be developed for ATM. If a customer enters an amount greater than the maximum permissible withdrawal amount, the ATM should display an error message, and it should not dispense any cash.

## Categorization of Requirements

Based on the target audience or subject matter, requirements can be classified into different types, as stated below:

* **User requirements:**They are written in natural language so that both customers can verify their requirements have been correctly identified
* **System requirements:**They are written involving technical terms and/or specifications, and are meant for the development or testing teams

Requirements can be classified into two groups based on what they describe:

* **Functional requirements (FRs):**These describe the functionality of a system -- how a system should react to a particular set of inputs and what should be the corresponding output.
* **Non-functional requirements (NFRs):**They are not directly related what functionalities are expected from the system. However, NFRs could typically define how the system should behave under certain situations. For example, a NFR could say that the system should work with 128MB RAM. Under such condition, a NFR could be more critical than a FR.

Non-functional requirements could be further classified into different types like:

* **Product requirements:** For example, a specification that the web application should use only plain HTML, and no frames
* **Performance requirements:** For example, the system should remain available 24x7
* **Organizational requirements:** The development process should comply to SEI CMM level 4

## Functional Requirements

### Identifying Functional Requirements

Given a problem statement, the functional requirements could be identified by focusing on the following points:

* Identify the high level functional requirements simply from the conceptual understanding of the problem. For example, a Library Management System, apart from anything else, should be able to issue and return books.
* Identify the cases where an end user gets some meaningful work done by using the system. For example, in a digital library a user might use the "Search Book" functionality to obtain information about the books of his interest.
* If we consider the system as a black box, there would be some inputs to it, and some output in return. This black box defines the functionalities of the system. For example, to search for a book, user gives title of the book as input and get the book details and location as the output.
* Any high level requirement identified could have different sub-requirements. For example, "Issue Book" module could behave differently for different class of users, or for a particular user who has issued the book thrice consecutively.

## Case Study

**A Library Information System**

As the size and capacity of the institute is increasing with the time, it has been proposed to develop a Library Information System (LIS) for the benefit of students and employees of the institute. LIS will enable the members to borrow a book (or return it) with ease while sitting at his desk/chamber. The system also enables a member to extend the date of his borrowing if no other booking for that particular book has been made. For the library staff, this system aids them to easily handle day-to-day book transactions. The librarian, who has administrative privileges and complete control over the system, can enter a new record into the system when a new book has been purchased, or remove a record in case any book is taken off the shelf. Any non-member is free to use this system to browse/search books online. However, issuing or returning books is restricted to valid users (members) of LIS only.

**Conclusion:** Thus we have studies how to identify and categories requirements from problem statement.

**Branch :- Computer Sci. & Engg. Class :- III Year**

**Subject :- Software Engineering Lab Sem :- VI**

**Student Manual**

**PRACTICAL NO 3**

**Aim: Estimation of Project Metrics.**

**Theory:**

A software project is not just about writing a few hundred lines of source code to achieve a particular objective. The scope of a software project is comparatively quite large, and such a project could take several years to complete. As in any other science and engineering discipline, one would be interested to measure how complex a project is. One of the major activities of the project planning phase, therefore, is to estimate various project parameters in order to take proper decisions.

**Some important project parameters that are estimated include:**

**Project size:** What would be the size of the code written say, in number of lines, files, modules?

**Cost:** How much would it cost to develop a software? A software may be just pieces of code, but one has to pay to the managers, developers, and other project personnel.

Duration: How long would it be before the software is delivered to the clients?

**Effort:**How much effort from the team members would be required to create the software?

**Two methods for estimating project metrics: COCOMO and Halstead's method.**

**COCOMO**

COCOMO (Constructive Cost Model) was proposed by Boehm. According to him, there could be three categories of software projects:

.

**1. Organic:** A development project is said to be of organic type, if

1)The project deals with developing a well understood application

2)The development team is small

3)The team members have prior experience in working with similar types of projects

**2. Semidetached**: A development project can be categorized as semidetached type, if

1)The team consists of some experienced as well as inexperienced staff

2)Team members may have some experience on the type of system to be developed

**3. Embedded:** Embedded type of development project are those, which

1)Aims to develop a software strongly related to machine hardware

2)Team size is usually large

Boehm suggested that estimation of project parameters should be done through three stages: Basic COCOMO, Intermediate COCOMO, and Detailed COCOMO.

1. **Basic COCOMO Model**

The basic COCOMO model helps to obtain a rough estimate of the project parameters. It estimates effort and time required for development in the following way:  
Effort = a \* (KLOC)b PM

Tdev = 2.5 \* (Effort)c Months

Where,

KLOC is the estimated size of the software expressed in Kilo lines of code

a, b, c are constants determined by the category of software project

Effort denotes the total effort required for the software development, expressed in person months (PMs)

Tdev denotes the estimated time required to develop the software (expressed in months)

The value of the constants a, b, c are given below:

| Software project | a | B | c |
| --- | --- | --- | --- |
| Organic | 2.4 | 1.05 | 0.38 |
| Semi-detached | 3.0 | 1.12 | 0.35 |
| Embedded | 3.6 | 1.20 | 0.32 |

1. **Intermediate COCOMO Model**

The basic COCOMO model considers that effort and development time depends only on the size of the software. However, in real life there are many other project parameters that influence the development process. The intermediate COCOMO take those other factors into consideration by defining a set of 15 cost drivers. Each of the 15 such attributes can be rated on a six-point scale ranging from "very low" to "extra high" in their relative order of importance.The product of effort multipliers of all the 15 attributes gives the Effort Adjustment Factor (EAF).

EAF is used to refine the estimates obtained by basic COCOMO as follows:

Effort\_corrected = Effort \* EAF

Dev\_corrected = 2.5 \* (Effort|\_corrected)c

1. **Detailed COCOMO Model**

Both the basic and intermediate COCOMO models consider a software to be a single homogeneous entity -- an assumption, which is rarely true. In fact, many real life applications are made up of several smaller sub-systems. The complete COCOMO model takes these factors into account to provide a far more accurate estimate of project metrics.

To illustrate this, consider a very popular distributed application: the ticket booking system of the Indian Railways. The ticket booking system has three main components:

Database, Graphical User Interface (GUI) and Networking facilities.

Among these, development of the GUI is considered as an organic project type; the database module could be considered as a semi-detached software. The networking module can be considered as an embedded software. To obtain a realistic cost, one should estimate the costs for each component separately, and then add it up.

* **Advantages of COCOMO**

COCOMO is a simple model, and should help one to understand the concept of project metrics estimation.

* **Drawbacks of COCOMO**

COCOMO uses KLOC, which is not a proper measure of a program's size. Indeed, estimating the size of a software is a difficult task.

**Case Study**

**A Library Information System**

Using COCOMO and based on the team size (small) and experience (high), the concerned project could be categorized as "organic". The experts, based on their prior experience, suggested that the project size could roughly be around 10 KLOC. This would serve as the basis for estimation of different project parameters using basic COCOMO, as shown below:

Effort = a \* (KLOC)b PM

Tdev = 2.5 \* (Effort)c Months

For organic category of project the values of a, b, c are 2.4, 1.05, 0.38 respectively.

Effort Adjustment Factor (EAF) by assigning appropriate weight to each of the following attributes.

| **Cost Drivers** | **Ratings** | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Very Low** | **Low** | **Nominal** | **High** | **Very High** | **Extra High** |
| **Product attributes** |  |  |  |  |  |  |
| Required software reliability | 0.75 | 0.88 | 1.00 | 1.15 | 1.40 |  |
| Size of database |  | 0.94 | 1.00 | 1.08 | 1.16 |  |
| Complexity of the product | 0.70 | 0.85 | 1.00 | 1.15 | 1.30 | 1.65 |
| **Hardware attributes** |  |  |  |  |  |  |
| Run-time performance constraints |  |  | 1.00 | 1.11 | 1.30 | 1.66 |
| Memory constraints |  |  | 1.00 | 1.06 | 1.21 | 1.56 |
| Volatility of the virtual machine environment |  | 0.87 | 1.00 | 1.15 | 1.30 |  |
| Required turnabout time |  | 0.87 | 1.00 | 1.07 | 1.15 |  |
| **Personnel attributes** |  |  |  |  |  |  |
| Analyst capability | 1.46 | 1.19 | 1.00 | 0.86 | 0.71 |  |
| Applications experience | 1.29 | 1.13 | 1.00 | 0.91 | 0.82 |  |
| Software engineer capability | 1.42 | 1.17 | 1.00 | 0.86 | 0.70 |  |
| Virtual machine experience | 1.21 | 1.10 | 1.00 | 0.90 |  |  |
| Programming language experience | 1.14 | 1.07 | 1.00 | 0.95 |  |  |
| **Project attributes** |  |  |  |  |  |  |
| Application of software engineering methods | 1.24 | 1.10 | 1.00 | 0.91 | 0.82 |  |
| Use of software tools | 1.24 | 1.10 | 1.00 | 0.91 | 0.83 |  |
| Required development schedule | 1.23 | 1.08 | 1.00 | 1.04 | 1.10 |  |

**Conclusion:** Thus we have studied project estimation techniques.

**Branch :- Computer Sci. & Engg. Class :- III Year**

**Subject :- Software Engineering Lab Sem :- VI**

**Teacher Manual**

**PRACTICAL NO 4**

**Aim:** [Modeling UML Use Case Diagrams and Capturing Use Case Scenarios](http://vlabs.iitkgp.ernet.in/se/3/)

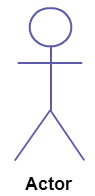
**Theory:**

1. **Use case diagrams**

Use case diagrams belong to the category of behavioural diagram of UML diagrams. Use case diagrams aim to present a graphical overview of the functionality provided by the system. It consists of a set of actions (referred to as use cases) that the concerned system can perform, one or more actors, and dependencies among them.

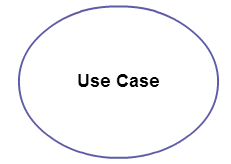
1. **Actor**

Actor in a use case diagram is any entity that performs a role in one given system. This could be a person, organization or an external system and usually drawn like skeleton shown below.



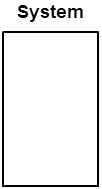
1. **Use Case**

A use case is simply a functionality provided by a system. A use case **represents a function or an action within the system**. It’s drawn as an oval and named with the function.



1. **Subject**

Subject is simply the system under consideration. Use cases apply to a subject. The system is used to define the scope of the use case and drawn as a rectangle.



1. **Graphical Representation**

An actor is represented by a stick figure and name of the actor is written below it. A use case is depicted by an ellipse and name of the use case is written inside it. The subject is shown by drawing a rectangle. Label for the system could be put inside it. Use cases are drawn inside the rectangle, and actors are drawn outside the rectangle, as shown in figure 01

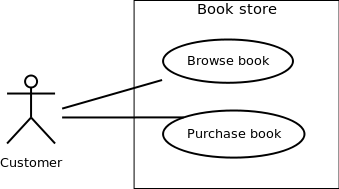


Figure - 01: A use case diagram for a book store

1. **Association between Actors and Use Cases**

A use case is triggered by an actor. Actors and use cases are connected through binary associations indicating that the two communicates through message passing.

An actor must be associated with at least one use case. Similarly, a given use case must be associated with at least one actor. Association among the actors are usually not shown. However, one can depict the class hierarchy among actors.

1. **Use Case Relationships**

Three types of relationships exist among use cases:

**Include relationship**

**Extend relationship**

**Use case generalization**

**Include Relationship**

Include relationships are used to depict common behaviour that are shared by multiple use cases. This could be considered analogous to writing functions in a program in order to avoid repetition of writing the same code. Such a function would be called from different points within the program.

Example

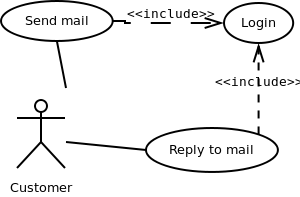


Figure - 02: Include relationship between use cases

Include relationship is depicted by a dashed arrow with a «include» stereotype from the including use case to the included use case.

**Extend Relationship**

Use case extensions are used to depict any variation to an existing use case. They are used to the specify the changes required when any assumption made by the existing use case becomes false.

Example:

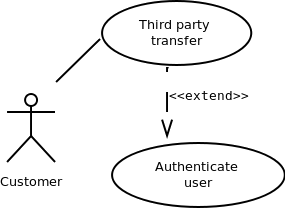


Figure - 03: Extend relationship between use cases

Extend relationship is depicted by a dashed arrow with a «extend» stereotype from the extending use case to the extended use case.

**Generalization Relationship**

Generalization relationship are used to represent the inheritance between use cases. A derived use case specializes some functionality it has already inherited from the base use case.

Example:

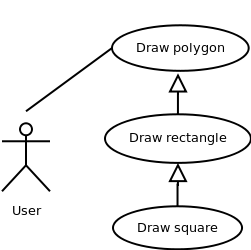


Figure - 04: Generalization relationship among use cases

Generalization relationship is depicted by a solid arrow from the specialized (derived) use case to the more generalized (base) use case.

1. **Identifying Actors**

Given a problem statement, the actors could be identified by asking the following questions :

* Who gets most of the benefits from the system? (The answer would lead to the identification of the primary actor)
* Who keeps the system working? (This will help to identify a list of potential users)
* What other software / hardware does the system interact with?
* Any interface (interaction) between the concerned system and any other system?

1. **Identifying Use cases**

Once the primary and secondary actors have been identified, we have to find out their goals i.e. what are the functionality they can obtain from the system. Any use case name should start with a verb like, "Check balance".

Guidelines for drawing Use Case diagrams

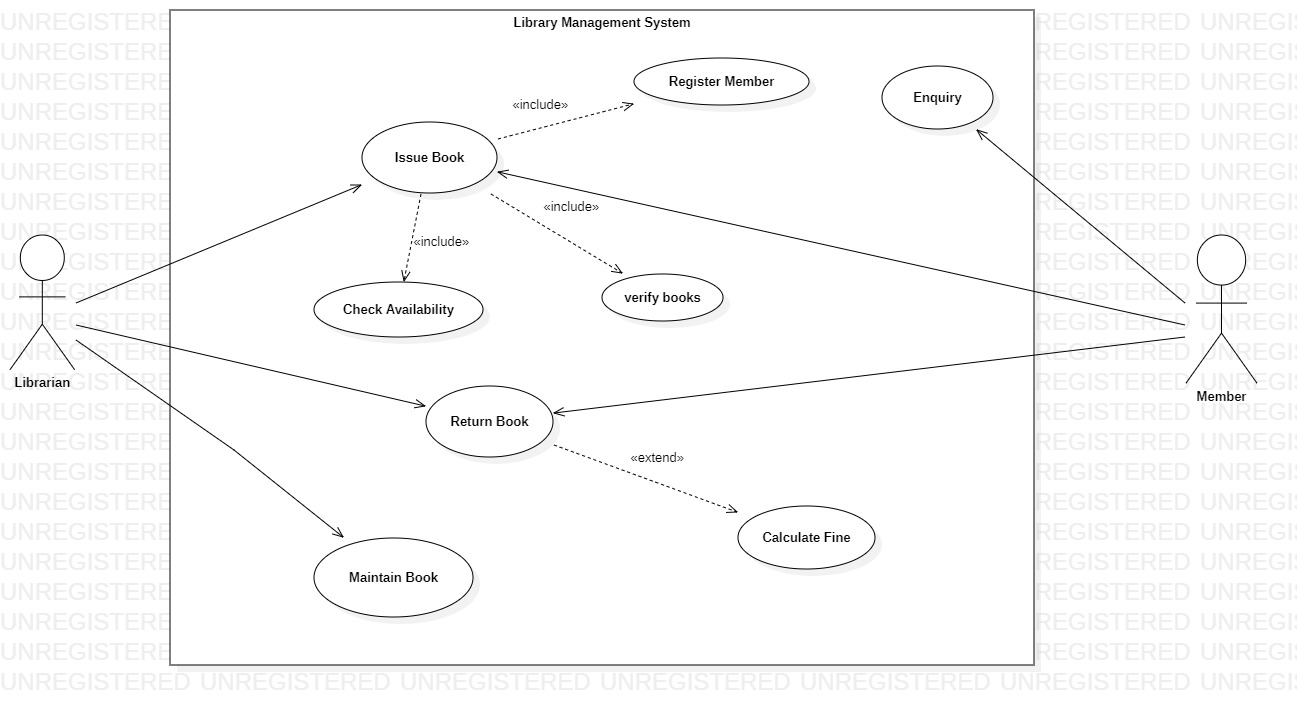
Following general guidelines could be kept in mind while trying to draw a use case diagram:

* Determine the system boundary
* Ensure that individual actors have well-defined purpose
* Use cases identified should let some meaningful work done by the actors
* Associate the actors and use cases -- there shouldn't be any actor or use case floating without any connection
* Use include relationship to encapsulate common behaviour among use cases , if any

**Case Study:**

* Draw a use case diagram for the following problem

Consider a library, where a member can perform two operations: issue book and return it. A book is issued to a member only after verifying his credentials. Draw a use case diagram for the problem.



**Conclusion:**Thus we have studied how to draw use case diagram for given case study.

**Branch :- Computer Sci. & Engg. Class :- III Year**

**Subject :- Software Engineering Lab Sem :- VI**

**Teacher Manual**

**PRACTICAL NO 5**

**Aim:** Modeling UML ER Diagrams and Sequence diagrams

**Theory:**

**Entity Relationship Model**

Entity-Relationship model is used to represent a logical design of a database to be created. In ER model, real world objects (or concepts) are abstracted as entities, and different possible associations among them are modeled as relationships.

**Entity Set and Relationship Set**

An entity set is a collection of all similar entities. For example, "Student" is an entity set that abstracts all students. Ram, John are specific entities belonging to this set. Similarly, a "Relationship" set is a set of similar relationships.

**Attributes of Entity**

Attributes are the characteristics describing any entity belonging to an entity set. Any entity in a set can be described by zero or more attributes.

For example, any student has got a name, age, an address. At any given time a student can study only at one school. In the school he would have a roll number, and of course a grade in which he studies. These data are the attributes of the entity set Student.

**Keys**

**One or more attribute(s) of an entity set can be used to define the following keys:**

* **Super key:** One or more attributes, which when taken together, helps to uniquely identify an entity in an entity set. For example, a school can have any number of students. However, if we know grade and roll number, then we can uniquely identify a student in that school.
* **Candidate key:** It is a minimal subset of a super key. In other words, a super key might contain extraneous attributes, which do not help in identifying an object uniquely. When such attributes are removed, the key formed so is called a candidate key.
* **Primary key:** A database might have more than one candidate key. Any candidate key chosen for a particular implementation of the database is called a primary key.
* **Prime attribute:** Any attribute taking part in a super key

**Weak Entity**

An entity set is said to be weak if it is dependent upon another entity set. A weak entity can't be uniquely identified only by it's attributes. In other words, it doesn't have a super key.

For example, consider a company that allows employees to have travel allowance for their immediate family. So, here we have two entity sets: employee and family, related by "Can claim for". However, family doesn't have a super key. Existence of a family is entirely dependent on the concerned employee. So, it is meaningful only with reference to employee.

**Entity Generalization and Specialization**

Once we have identified the entity sets, we might find some similarities among them. For example, multiple person interacts with a banking system. Most of them are customers, and rest employees or other service providers. Here, customers, employees are persons, but with certain specializations. Or in other way, person is the generalized form of customer and employee entity sets.

ER model uses the "ISA" hierarchy to depict specialization (and thus, generalization).

**Mapping Cardinalities**

One of the main tasks of ER modeling is to associate different entity sets. Let's consider two entity sets E1 and E2 associated by a relationship set R. Based on the number of entities in E1 and E2 are associated with, we can have the following four type of mappings:

* One to one: An entity in E1 is related to at most a single entity in E2, and vice versa
* One to many: An entity in E1 could be related to zero or more entities in E2. Any entity in E2 could be related to at most a single entity in E1.
* Many to one: Zero or more number of entities in E1 could be associated to a single entity in E2. However, an entity in E2 could be related to at most one entity in E1.
* Many to many: Any number of entities could be related to any number of entities in E2, including zero, and vice versa

**ER Diagram**

From a given problem statement we identify the possible entity sets, their attributes, and relationships among different entity sets. Once we have these information, we represent them pictorially, called an entity-relationship (ER) diagram.

**Graphical Notations for ER Diagram:**

| **Term** | **Notation** | **Remarks** |
| --- | --- | --- |
| **Entity set** | **IMG_256** | **Name of the set is written inside the rectangle** |
| **Attribute** | **IMG_257** | **Name of the attribute is written inside the ellipse** |
| **Entity with attributes** | **IMG_258** | **Roll is the primary key; denoted with an underline** |
| **Weak entity set** | **IMG_259** |  |
| **Relationship set** | **IMG_260** | **Name of the relationship is written inside the diamond** |
| **Related enity sets** | **IMG_261** |  |
| **Relationship cardinality** | **IMG_262** | **A person can own zero or more cars but no two persons can own the same car** |
| **Relationship with weak entity set** | **IMG_263** |  |

**Importance of ER modeling**

Figure - 01 shows the different steps involved in implementation of a (relational) database.

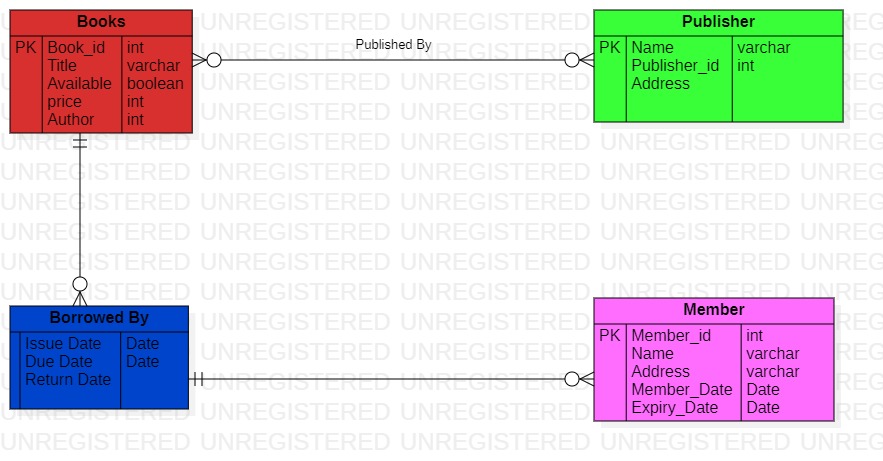
IMG_264

Figure - 01: Steps to implement a RDBMS

Given a problem statement, the first step is to identify the entities, attributes and relationships. We represent them using an ER diagram. Using this ER diagram, table structures are created, along with required constraints. Finally, these tables are normalized in order to remove redundancy and maintain data integrity. Thus, to have data stored efficiently, the ER diagram is to be drawn as much detailed and accurate as possible.

**Case Study:**

Draw ER diagram for the Library Management System.



**Sequence diagram**

It represents the behavioral aspects of a system. Sequence diagram shows the interactions between the objects by means of passing messages from one object to another with respect to time in a system.

**Elements in sequence diagram**

Sequence diagram contains the objects of a system and their life-line bar and the messages passing between them.

**Object**

Objects appear at the top portion of sequence diagram. Object is shown in a rectangle box. Name of object precedes a colon ‘:’ and the class name, from which the object is instantiated. The whole string is underlined and appears in a rectangle box. Also, we may use only class name or only instance name.

Objects which are created at the time of execution of use case and are involved in message passing , are appear in diagram, at the point of their creation.

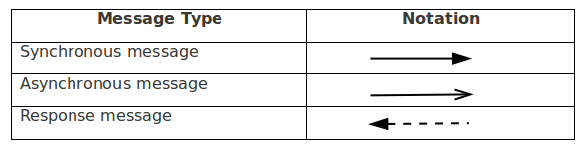
**Life-line bar**

A down-ward vertical line from object-box is shown as the life-line of the object. A rectangle bar on life-line indicates that it is active at that point of time.

**Messages**

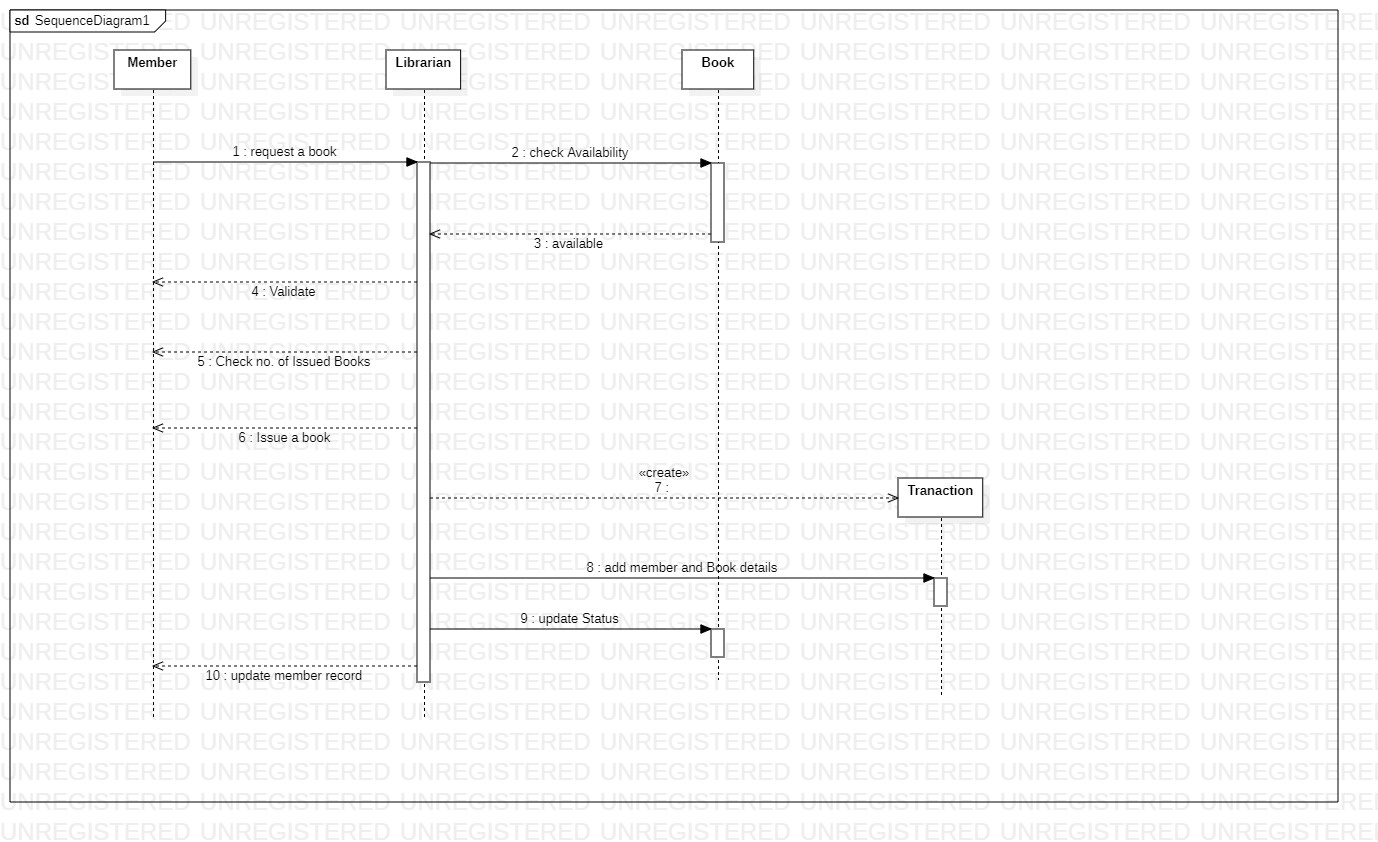
Messages are shown as an arrow from the life-line of sender object to the life-line of receiver object and labeled with the message name. Chronological order of the messages passing throughout the objects’ life-line show the sequence in which they occur. There may exist some different types of messages :

* **Synchronous messages**:Receiver start processing the message after receiving it and sender needs to wait until it is made. A straight arrow with close and fill arrow-head from sender life-line bar to receiver end, represent a synchronous message.
* **Asynchronous messages**:For asynchronous message sender needs not to wait for the receiver to process the message. A function call that creates thread can be represented as an asynchronous message in sequence diagram. A straight arrow with open arrow-head from sender life-line bar to receiver end, represent an asynchronous message.
* **Return message:**For a function call when we need to return a value to the object, from which it was called, then we use return message. But, it is optional, and we are using it when we are going to model our system in much detail. A dashed arrow with open arrow-head from sender life-line bar to receiver end, represent that message.
* **Response message:**One object can send a message to self. We use this message when we need to show the interaction between the same object.



**Case Study:**

Draw Sequence diagram for the Library Management System.



**Conclusion**: Thus we have studied, Modeling UML Class Diagrams and Sequence diagrams

**Branch :- Computer Sci. & Engg. Class :- III Year**

**Subject :- Software Engineering Lab Sem :- VI**

**Teacher Manual**

**PRACTICAL NO 6**

**Aim:** [Statechart and Activity Modeling](http://vlabs.iitkgp.ernet.in/se/6/)

**Theory:**

1. **Statechart Diagrams**

A statechart diagram is a pictorial representation of a system, with all it's states, and different events that lead transition from one state to another.

To illustrate this, consider a computer. Some possible states that it could have are: running, shutdown, hibernate. A transition from running state to shutdown state occur when user presses the "Power off" switch, or clicks on the "Shut down" button as displayed by the OS. Here, clicking on the shutdown button, or pressing the power off switch act as external events causing the transition.

Statechart diagrams are normally drawn to model the behaviour of a complex system. For simple systems this is optional.

**Building Blocks of a Statechart Diagram**

**State**

A state is any "distinct" stage that an object (system) passes through in it's lifetime. An object remains in a given state for finite time until "something" happens, which makes it to move to another state.  All such states can be broadly categorized into following three types:

**Initial:** The state in which an object remain when created

**Final:** The state from which an object do not move to any other state [optional]

**Intermediate:** Any state, which is neither initial, nor final

As shown in figure-01, an initial state is represented by a circle filled with black. An intermediate state is depicted by a rectangle with rounded corners. A final state is represented by a unfilled circle with an inner black-filled circle.

IMG_256

**Transition**

Transition is movement from one state to another state in response to an external stimulus (or any internal event). A transition is represented by a solid arrow from the current state to the next state.

1. **Activity Diagrams**

Activity diagrams fall under the category of behavioural diagrams in Unified Modeling Language. It is a high level diagram used to visually represent the flow of control in a system. It has similarities with traditional flow charts. However, it is more powerful than a simple flow chart since it can represent various other concepts like concurrent activities, their joining, and so on. These kind of diagrams are suitable for confirming the logic to be implemented with the business users. These diagrams are typically used when the business logic is complex. In simple scenarios it can be avoided entirely.

**Components of an Activity Diagram**

Below we describe the building blocks of an activity diagram.

**Activity**

An activity denotes a particular action taken in the logical flow of control. An activity is represented with a rounded rectangle.

There are two special type of activity nodes: initial and final. They are represented with a filled circle, and a filled in circle with a border respectively. Initial node represents the starting point of a flow in an activity diagram. There could be multiple initial nodes.

A final node represents the end point of all activities. Like an initial node, there could be multiple final nodes. Any transition reaching a final node would stop all activities.

**Flow**

A flow (also termed as edge, or transition) is represented with a directed arrow. This is used to depict transfer of control from one activity to another.

**Decision**

A decision node, represented with a diamond, is a point where a single flow enters and two or more flows leave. The control flow can follow only one of the outgoing paths. The outgoing edges often have guard conditions indicating true-false or if-then-else conditions. **Merge**

This is represented with a diamond shape, with two or more flows entering, and a single flow leaving out. A merge node represents the point where at least a single control should reach before further processing could continue.

**Fork**

Fork is a point where parallel activities begin. For example, when a student has been registered with a college, he can in parallel apply for student ID card and library card. A fork is graphically depicted with a black bar, with a single flow entering and multiple flows leaving out.

**Join**

A join is depicted with a black bar, with multiple input flows, but a single output flow. Physically it represents the synchronization of all concurrent activities. Unlike a merge, in case of a join all of the incoming controls must be completed before any further progress could be made.

**Note**

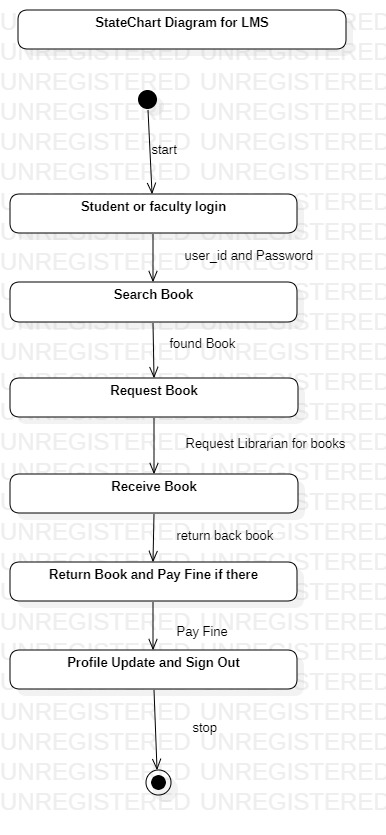
UML allows attaching a note to different components of a diagram to present some textual information. The information could simply be a comment or may be some constraint.

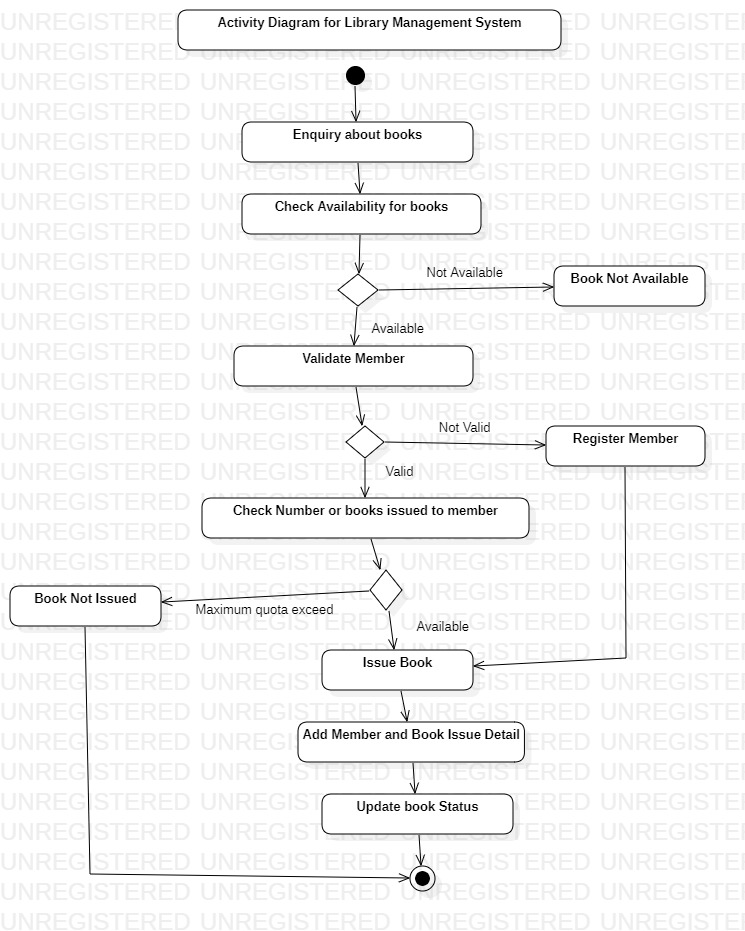
The following table shows commonly used components with a typical activity diagram.

| **Component** | **Graphical Notation** |
| --- | --- |
| Activity | IMG_256 |
| Flow | IMG_257 |
| Decision | IMG_258 |
| Merge | IMG_259 |
| Fork | IMG_260 |
| Join | IMG_261 |
| Note | IMG_262 |

Table-01: Typical components used in an activity diagram

**Case Study:** Draw Statechart diagram and activity diagram for Library Management System.





**Conclusion:** Thus we have studies how to draw statechart diagram and activity diagram.

**Branch :- Computer Sci. & Engg. Class :- III Year**

**Subject :- Software Engineering Lab Sem :- VI**

**Teacher Manual**

**PRACTICAL NO 7**

**Aim:** Modeling Data Flow Diagram

**Theory:**

**Data Flow Diagram**

DFD provides the functional overview of a system. A Data Flow Diagram (DFD) is a traditional visual representation of the information flows within a system. DFD shows the external entities from which data flows into the process and also the other flows of data within a system. It also includes the transformations of data flow by the process and the data stores to read or write a data.

**Graphical notations for Data Flow Diagram**

| Term | Notation | Remarks |
| --- | --- | --- |
| External entity | IMG_256 | Name of the external entity is written inside the rectangle |
| Process | IMG_257 | Name of the process is written inside the circle |
| Data store | IMG_258 | A left-right open rectangle is denoted as data store; name of the data store is written inside the shape |
| Data flow | IMG_259 | Data flow is represented by a directed arc with its data name |

**Explanation of Symbols used in DFD**

**Process:** Processes are represented by circle. The name of the process is written into the circle. The name of the process is usually given in such a way that represents the functionality of the process.

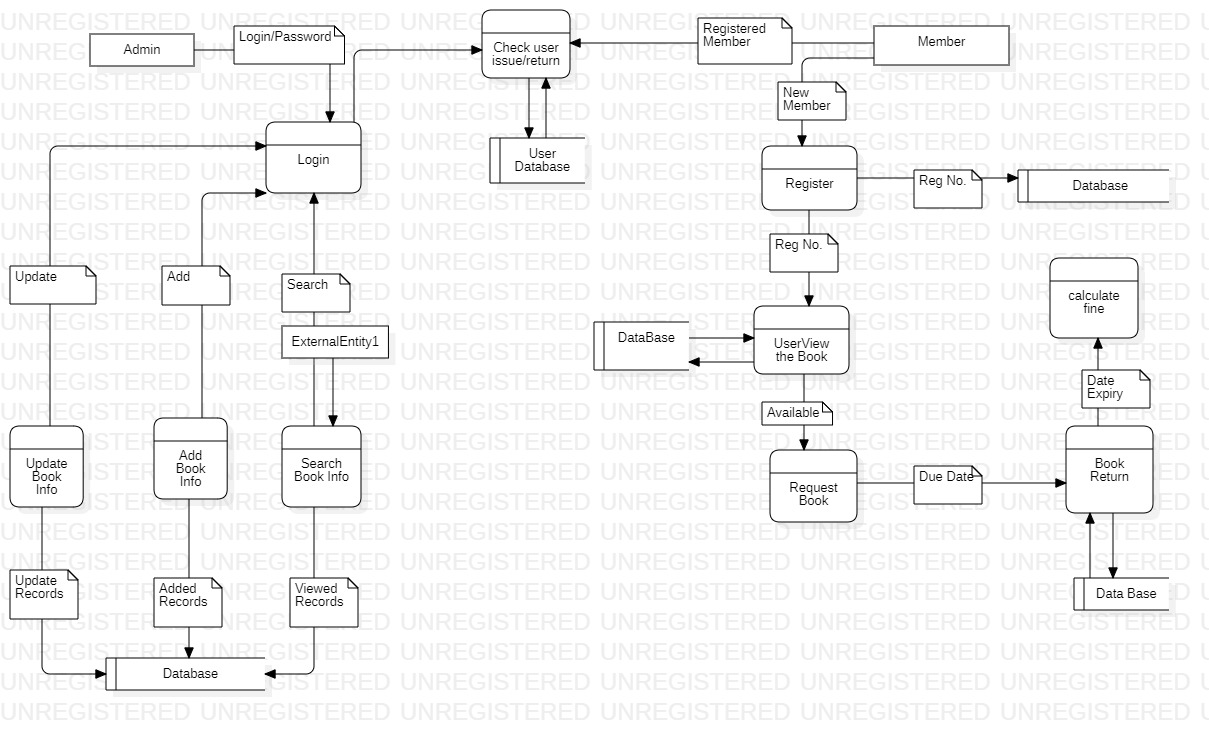
**External entity:** External entities are represented by a rectangle and the name of the external entity is written into the shape. These send data to be processed and again receive the processed data.

**Data store:** Data stores are represented by a left-right open rectangle. Name of the data store is written in between two horizontal lines of the open rectangle. Data stores are used as repositories from which data can be flown in or flown out to or from a process.

**Data flow:** Data flows are shown as a directed edge between two components of a Data Flow Diagram. Data can flow from external entity to process, data store to process, in between two processes and vice-versa.

**Data Flow Diagram of Library Management System:**

**Case Study:** Draw DFD diagram for Library Management System.



**Conclusion**: Thus we have studied, Modeling Data Flow Diagram.

**Branch :- Computer Sci. & Engg. Class :- III Year**

**Subject :- Software Engineering Lab Sem :- VI**

**Teacher Manual**

**PRACTICAL NO 8**

**Aim:** Study of various testing methods

**Theory:**

**Software Testing**

Testing software is an important part of the development life cycle of software. It is an expensive activity. Hence, appropriate testing methods are necessary for ensuring the reliability of a program. According to the ANSI/IEEE 1059 standard, the definition of testing is the process of analyzing a software item, to detect the differences between existing and required conditions i.e. defects/errors/bugs and to evaluate the features of the software item.

The purpose of testing is to verify and validate software and to find the defects present in software. The purpose of finding those problems is to get them fixed.

* Verification is the checking or we can say the testing of software for consistency and conformance by evaluating the results against pre-specified requirements.
* Validation looks at the systems correctness, i.e. the process of checking that what has been specified is what the user actually wanted.
* Defect is a variance between the expected and actual result. The defect’s ultimate source may be traced to a fault introduced in the specification, design, or development (coding) phases.

**Testing Frameworks**

Following are the different testing frameworks:

* **jUnit** - for Java unit test
* **Selenium** - is a suite of tools for automating web applications for software testing purposes, plugin for Firefox
* **HP QC** - is the HP Web-based test management tool. It familiarizes with the process of defining releases, specifying requirements, planning tests, executing tests, tracking defects, alerting on changes, and analyzing results. It also shows how to customize project
* **IBM Rational** - Rational software has a solution to support business sector for designing, implementing and testing software

**Need for Software Testing**

There are many reasons for why we should test software, such as:

* Software testing identifies the software faults. The removal of faults helps reduce the number of system failures. Reducing failures improves the reliability and the quality of the systems.
* Software testing can also improves the other system qualities such as maintainability, usability, and testability.
* In order to meet the condition that the last few years of the 20th century systems had to be shown to be free from the ‘millennium bug’.
* In order to meet the different legal requirements.
* In order to meet industry specific standards such as the Aerospace, Missile and Railway Signaling standards.

**Test Cases and Test Suite**

A test case describes an input description and an expected output descriptions. Input are of two types: preconditions (circumstances that hold prior to test case execution) and the actual inputs that are identified by some testing methods. The set of test cases is called a test suite. We may have a test suite of all possible test cases.

**Types of Software Testing**

Testing is done in every stage of software development life cycle, but the testing done at each level of software development is different in nature and has different objectives. There are different types of testing, such as stress testing, volume testing, configuration testing, compatibility testing, recovery testing, maintenance testing, documentation testing, and usability testing. Software testing are mainly of following types:

* **Black box testing:** This is also known as functional testing , where the test cases are designed based on input output values only. There are many types of Black Box Testing but following are the prominent ones.

- Equivalence class partitioning: In this approach, the domain of input values to a program is divided into a set of equivalence classes. e.g. Consider a software program that computes whether an integer number is even or not that is in the range of 0 to 10. Determine the equivalence class test suite. There are three equivalence classes for this program. - The set of negative integer - The integers in the range 0 to 10 - The integer larger than 10

- Boundary value analysis : In this approach, while designing the test cases, the values at boundaries of different equivalence classes are taken into consideration. e.g. In the above given example as in equivalence class partitioning, a boundary values based test suite is { 0, -1, 10, 11 }

* **White box testing:** It is also known as structural testing. In this testing, test cases are designed on the basis of examination of the code.This testing is performed based on the knowledge of how the system is implemented. It includes analyzing data flow, control flow, information flow, coding practices, exception and error handling within the system, to test the intended and unintended software behavior. White box testing can be performed to validate whether code implementation follows intended design, to validate implemented security functionality, and to uncover exploitable vulnerabilities.This testing requires access to the source code. Though white box testing can be performed any time in the life cycle after the code is developed, but it is a good practice to perform white box testing during the unit testing phase.

1. **Unit Testing**
2. **Integration Testing**
3. **System Testing**
4. **Unit Testing**

Unit testing is done at the lowest level. It tests the basic unit of software that is the smallest testable piece of software. The individual component or unit of a program is tested in unit testing. Unit testing are of two types.

1. **Integration Testing**

Integration testing is performed when two or more tested units are combined into a larger structure. The main objective of this testing is to check whether the different modules of a program interface with each other properly or not. This testing is mainly of two types:

* Top-down approach
* Bottom-up approach

In bottom-up approach, each subsystem is tested separately and then the full system is tested. But the top-down integration testing starts with the main routine and one or two subordinate routines in the system. After the top-level ‘skeleton’ has been tested, the immediately subroutines of the ‘skeleton’ are combined with it and tested.

1. **System Testing**

System testing tends to affirm the end-to-end quality of the entire system. System testing is often based on the functional / requirement specification of the system. Non-functional quality attributes, such as reliability, security, and maintainability are also checked. There are three types of system testing

* Alpha testing is done by the developers who develop the software. This testing is also done by the client or an outsider with the presence of developer or we can say tester.
* Beta testing is done by very few numbers of end users before the delivery, where the change requests are fixed, if the user gives any feedback or reports any type of defect.
* User Acceptance testing is also another level of the system testing process where the system is tested for acceptability. This test evaluates the system's compliance with the client requirements and assess whether it is acceptable for software delivery

An error correction may introduce new errors. Therefore, after every round of error-fixing, another testing is carried out, i.e. called regression testing. Regression testing does not belong to either unit testing, integration testing, or system testing, instead, it is a separate dimension to these three forms of testing.

**Regression Testing**

The purpose of regression testing is to ensure that bug fixes and new functionality introduced in software do not adversely affect the unmodified parts of the program. Regression testing is an important activity at both testing and maintenance phases. When a piece of software is modified, it is necessary to ensure that the quality of the software is preserved. To this end, regression testing is to retest the software using the test cases selected from the original test suite.

**Conclusion:** Thus we have studied various software Testing Methods.