Unit-1

Software Development Process

❖ SOFTWARE

(IEEE Definition of software): Software is a "Collection of computer programs, procedures, rules, associated documents and concerned data with the operation of data processing system."

It also includes representations of pictorial, video, and audio information.

Software is divided into two broad categories:

System software: It is responsible for controlling and integrating the

hardware components of a system.

Hence, the software and users can work with them. It also controls the peripherals of computer like monitors, printers, storage devices etc.

Example: Operating system

Application software: It is used to accomplish some specific tasks. It should be collection of small programs.
It is a program or group of programs generally designed for end users.

Example: Microsoft word, Excel, Railway reservation system etc.

Computer hardware is built from peripherals, device or components, while computer software is logical rather than physical.

Software has following distinct characteristics.

- **Software characteristics:**

Different software characteristics decide whether the software is good or bad. These attributes reflect the quality of a software product. And these are depended on the application of the software also.

For example, a banking system must be secure while a telephone switching system must be reliable.

Following are the characteristics of good software: (Qualities of good software).

Understandability: Software should be easy to understand, even to novice users. It should be efficient to use.

Cost: Software should be cost effective as per its usage.

Maintainability: Software should be easily maintainable and modifiable in future.

Modularity: Software should have modular approach so it can be handled easily for testing.

Functionality: Software should be functionally capable to meet user's requirements.

Reliability: It should have the capability to provide failure-free service.

Portability: Software should have the capability to be adapted for different environments.

Correctness: Software should be correct as per its requirements.

Documentation: Software should be properly documented so that we can re-refer it in future.

Reusability: It should be reusable, or its code or logic should be reusable in future.

Interoperability: Software should be able to communicate with various devices using standard bus structure and protocol.

Verifiability: Software should be verifiable with its properties and functionalities with its planning and analysis done in previous phase.

Along with these characteristics, software has some special characteristics which are explained below.

- **Software doesn't wear out:**

This can be well understood with the help of figure given below.

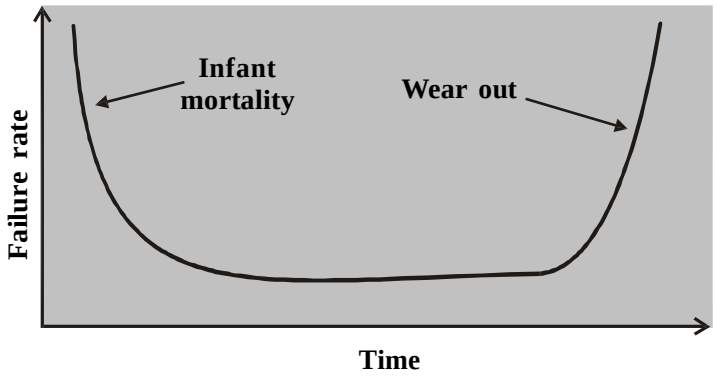


Figure: hardware failure curve

In above figure, the relationship between time and failure called "bath-tub curve". It indicates that hardware has relatively high failure rates early in its life, defects are corrected and the failure rate drops to a steady-state level for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the effects of dust, vibration, abuse, temperature extremes, and many other environmental factors. So simply, we can say hardware begins to wear out.

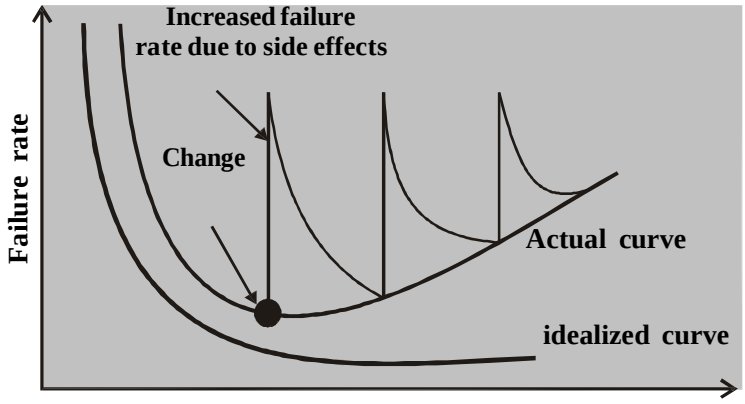


Figure: Software failure curve

Above figure shows the software failure rate.

- Software is not highly affected by environmental effects. The "idealized curve" shows software failure.
- In the early stage, due to lot many errors, software could have high failure. But it becomes reliable as time passes instead of wearing out. Once software is made it has a longer life span that we can see in idealized curve.
- In actual curve (above figure), we can see that software may have increased failure rate as it may become old as and when the new development environment changes. Spike in the curve is due to chance of maintenance and side effects.

- Software may be retired due to new requirements, new expectations, new technology new technologies etc. Hence, software doesn't wear out, but it may get worse.

▪ **Software is engineered, not manufactured like hardware:**

Once a product is manufactured, it is not easy to modify or change it. While in case of software we can easily change or modify it for later use. Even making multiple copies of software is a very easy task rather it is much more tuff in case of hardware.

In hardware, costing is due to assembly of raw material and other processing expenses while in software development, no assembly needed like hardware. Hence, software is not manufactured as it is developed or it is engineered.

▪ **Reusability of components:**

- If we assemble hardware, we will have every part and component from different vendors and then we may produce a finished product.
- In case of software, every project is a new project and we have to start from scratch and design every unit of software product. Huge number of efforts required to develop a software system. So, building a standard code or design is very useful for making many new projects. These codes are reusable.
- We can reuse software codes, modules or any logical components in any other related software projects. (For example, we prepared a payroll system for any organization. We can reuse some of the logical components while we are developing the related types of payroll systems for any other company or organizations.)
- Generally, GUI (graphical user interface) software is built using reusable components.

▪ **Software is flexible for custom built:**

- A software program can be developed to do anything. Any kind of change needed in software can be done easily.
- A software program or product can be built on user requirements basis or custom built. Even the developed software product can also be changed as per the user demands.
- We can say software is very much flexible for custom built rather than hardware.
- Hence, now a days industries are moving toward component-based assembly, software continues to be custom built.

❖ SOFTWARE ENGINEERING

- **Definition:**

Software Engineering discipline began since 5 decade and provides solution to software crisis.

Software crisis is the difficulty of writing useful and efficient computer program in the required time.

- ***Software Engineering is an engineering discipline that delivers high quality software at agreed cost & in planned schedule.***
- Three main aspects of Software Engineering are:
  - Provide quality product
  - Expected cost
  - Complete work on agreed schedule

(IEEE Definition) Software Engineering is the application of a systematic, disciplined and quantifiable approach to the development, operation and maintenance of software.

- **Software Engineering - A layered approach:**

Software engineering can be viewed as a layered technology.

It contains process, methods and tools that enable software product to be built in a timely manner.

Figure: Software Engineering Layers

As we can see in the above figure, this layered approach contains mainly four layers.

1. **A quality focus layer:**
   - Software engineering mainly focuses on quality product.
   - It checks whether the output meets with its requirement specifications or not.
   - Every organization should maintain its total quality management (TQM).
   - This layer supports software engineering.
2. **Process layer:**
   - Software process is a set of activities together if ordered and performed properly, the desired result would be produced.
   - Main objective of this layer is to develop software in time.
   - This layer is the heart of software engineering.
   - It holds all the technology layers together like GLUE.
   - It is also working as foundation layer.
   - It defines the framework activities.
3. **Method layer:**
   - It provides technical knowledge for developing software. It describes 'how-to' build software product.
   - It creates software engineering environment to software product using CASE tools. (CASE tools combines software, hardware and software engineering database).
   - This layer includes requirements analysis, design, program construction, testing, and support.
4. **Tools layer:**
   - It provides support to below layers using automated or semi-automated tools.
   - Due to this layer, process is executed in proper manner.

- **Need of software engineering:**

The reasons why software engineering is needed to develop software products are given below:

- To help developers to obtain high quality software product.
- To develop the product in appropriate manner using life cycle models.
- To acquire skills to develop large programs.

- To acquire skills to be a better programmer.
- To provide a software product in a timely manner.
- To provide a quality software product.
- To provide a software product at an agreed cost.
- To develop ability to solve complex programming problems.

Also learn techniques of specification, design, user interface development, testing, project management, etc.

❖ SOFTWARE DEVELOPMENT

- Software development is the process of developing software through successive phases in an ordered way.
- This process includes not only the actual writing of code but also the preparation of requirements and objectives, the design of what is to be coded, and confirmation that what is developed has met objectives or not.
- In other words, Software development is the computer programming, documenting, testing, and bug fixing involved in creating and maintaining applications and frameworks involved in a software release life cycle and resulting in a software product.
- Software can be developed for a variety of purposes.
- Three most common being for software development:
  - To meet specific needs of specific clients. (i.e., banking software)
  - To meet the need of some set of potential users. (i.e., Facebook or WhatsApp etc.)
  - To develop for personal use. (i.e., software that is built for individual use)

Software development process is a set of steps that a software program goes through when developed.

General phases of software development are:

Requirements → Analysis → Design → Implementation →
Testing / Verification → Documentation → Maintenance

- First in the software development process, the requirements phase outlines the goals of what the program will be capable of doing.
- Next, the design phase covers how the program is going to be created, who will be doing what etc.
- The implementation phase is where the programmers and other designers start work on the program.
- Testing and verification step can begin to help verify the program has no errors. During the testing phase, problems found are fixed, until the program meets the company's quality controls.
- After the program's development, the documentation phase begins to describe how to use the program or product.
- Finally, maintaining (updating) the program must continue for several years after the initial release.

Importance of software development:

- Software is important to make the hardware working.
- Software is important to build up security where it needs to be done, such as, in banks, money transaction etc.
- Software is important to make a task easier, such as, distribution of products, products information etc.
- Software is important to use the power of computer or working efficiency to perform those tasks which cannot be done or controlled by human.

- Shortly, it can be said that software is important to make things easier, faster, more reliable and safer.

Generic view of software engineering

- The work associated with software engineering can be categorized into three generic phases → Definition phase, Development phase and Support phase.

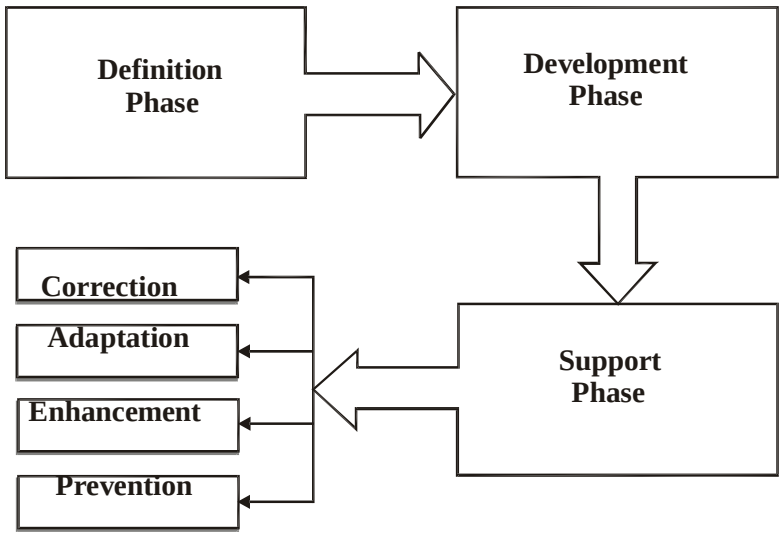


Figure: Software generic view

1. **The Definition Phase:**

- It focuses on *"what part"*. That is, during definition, the software engineer attempts to identify what information is to be processed, what function and performance are desired, what system behaviour can be expected, what interfaces are to be established, what design constraints exist, and what validation criteria are required to define a successful system.
- The key requirements of the system and the software are identified.
- Three main activities performed during this phase: system or information engineering, software project planning and requirement analysis.

2. **The development phase:**

- It focuses on *"how part"* of the development. During development a software engineer attempts to define how data are to be structured, how function is to be implemented within a software architecture, how interfaces are to be characterized, how the design will be translated into a programming language, and how testing will be performed.
- Main activities are performed under this phase are: software design, code generation and software testing.

3. **The support phase:**

- The support phase focuses on "change" associated with error correction.
- Four types of change are encountered during the support phase:
  - → **Correction:** corrective maintenance changes the software to correct defects.
  - → **Adaption:** Adaptive maintenance results in modification to the software to accommodate changes to its external environment.
  - → **Enhancement/Perfection:** Perfective maintenance extends the software beyond its original functional requirements.

→ **Prevention:** Preventive maintenance to enable the software to serve the needs of its end users.

The definition phase focuses on '**what**'
The development phase focuses on '**how**'
The support phase focuses on '**change**'

❖ GENERIC (NOT SPECIFIC TO A GROUP) FRAMEWORK ACTIVITIES

A software framework provides a standard way to build deploy software product. And a software process is the set of activities and associated results that produce a software product.

Any standard software process model would primarily consist of two types of activities: **A set of framework activities,** which are always applicable to all the projects and **A set of umbrella activities** which are non SDLC activities that are applicable throughout the process.

It provides common process framework for all projects.

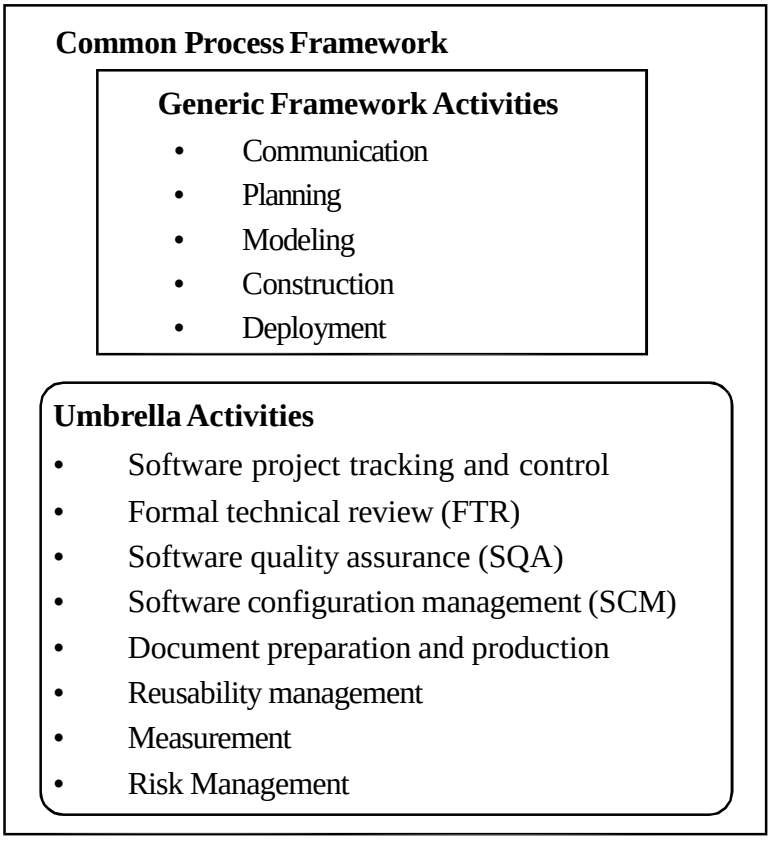


Figure: Generic framework and Umbrella activities

The Adaptable Process Model (APM) defines the following set of framework activities.

Communication

The software development starts with the communication between customer and developer.

Planning

It consists of complete estimation, scheduling for project development and tracking.

Modeling

Modeling consists of complete requirement analysis and the design of the project like algorithm, flowchart etc.

Construction

- Construction consists of code generation and the testing part.
- Coding part implements the design details using an appropriate programming language.
- Testing is to check whether the flow of coding is correct or not.
- Testing also check that the program provides desired output.

Deployment

- Deployment step consists of delivering the product to the customer and take feedback from them.
- If the customer wants some corrections or demands for the additional capabilities, then the change is required for improvement in the quality of the software.

- **Umbrella activities:**

- The phases and related steps of the generic view of software engineering are complemented by a number of umbrella activities.
- Umbrella activities are performed throughout the process.
- These activities are independent of any framework activity.

Typical activities in this category include:

1. **Software project tracking and control**

   When project tracking and controlling done then software engineering tasks will enable to get the job done on time.

2. **Formal technical review (FTR)**

   This includes reviewing the techniques that has been used in the project.

3. **Software quality assurance (SQA)**

   This is very important to ensure the quality measurement of each part of software being developed.

4. **Software configuration management (SCM)**

   SCM is a set of activities designed to control changes made by identifying the work products that are likely to change, establishing relationships among them.

5. **Document preparation and production**

   All the project planning and other activities should be hardly copied and the production gets started here.

6. **Reusability management**

   This includes the backing up of each part of the software project they can be corrected or any kind of support can be given to them later to update or upgrade the software at user/time demand.

7. **Measurement (estimation)**

   This will include all the measurement or estimation of every aspects of the software project like: time estimation, cost estimation etc.

8. Risk management

- As we know that 'tomorrow's problem is today's risk'. Risk management is very important activity for any type of software development.
- It identifies potential problems and deal with them when they are easier to handle before they become critical.
- Risk management allows early identification of risks and provide management decisions to the solutions, and improve quality of the product.

❖ SOFTWARE DEVELOPMENT MODELS / LIFE CYCLE MODELS

- Every system has a life cycle. It begins when a problem is recognized, after then system is developed, grows until maturity and then maintenance needed due to change in the nature of the system. So it died and new system or replacement of it taken place. (This can be shown in below figure)
- Software process models describe the sequence of activities needed to develop a software product.

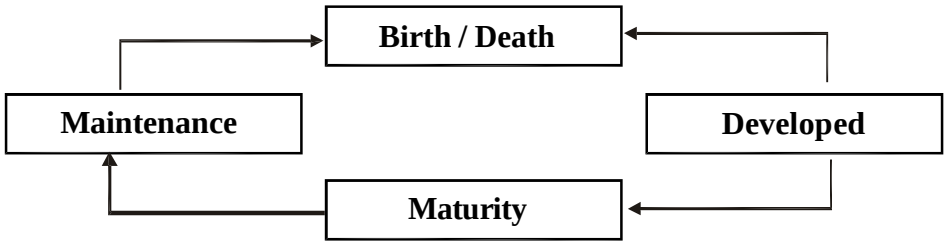


Figure: System Life Cycle

- The goal of the software development process is to produce high quality software product.
- As per IEEE Standards, software life cycle is: "the period of time that starts when software product is conceived and ends when the product is no longer available for use."
- A software life cycle model is also called a Software Development Life Cycle (SDLC). More suitable definition of SDLC is given below:

> **Software Development Life Cycle (SDLC) is a process used by the software industry to design, develop and test high quality software, which meets exact customer's requirements and completion within time and cost estimation.**

- Software life cycle is the series of identifiable stages that a software product undergoes during its lifetime.
- Software life cycle model (process model) is a descriptive and diagrammatic representation of the software life cycle.
- A life cycle model represents all the activities required to make a software product transit through its life cycle phases.
- General stages of software development life cycle are → feasibility study, requirement analysis and specification, design, coding, testing and maintenance.

Need of life cycle models:

- Process models provide generic guidelines for developing a suitable process for a project.
- It provides improvement and guarantee of quality product.
- Without using of a particular life cycle model, the development of a software product would not be in a systematic and disciplined manner.
- Provide monitoring the progress of the project-to-project managers.

- A software life cycle model defines entry and exit criteria for every phase.
- These criteria used to chart the progress of the project.
- A phase can begin only when the corresponding entry criteria are satisfied.
- If entry and exit criteria for various phases are satisfied, it's easy to monitor the progress of the project.
- The documentation of life cycle models enhances the understanding between developers and client.

Different Software Development Life Cycle (SDLC) models OR Software Development Models:

Many life cycle models have been proposed so far. Each of them has some advantages as well as some disadvantages. A few important and commonly used life cycle models are as follows:

(1) Classical waterfall model
(2) Iterative waterfall model
(3) Incremental Evolutionary model
(4) RAD model
(5) Prototype model
(6) Spiral model

(1) Classical Waterfall model:

- This model was originally proposed by Royce (1970). It is also called 'linear sequential life cycle model'.
- It is also called 'traditional waterfall model' or 'conventional waterfall model'.
- It's just a theoretical way of developing software All other life cycle models are essentially derived from it.
- This model breaks down the life cycle into set of phases like:
  - Feasibility study
  - Requirements analysis and specification
  - Design
  - Coding and unit testing
  - Integration and system testing
  - Maintenance

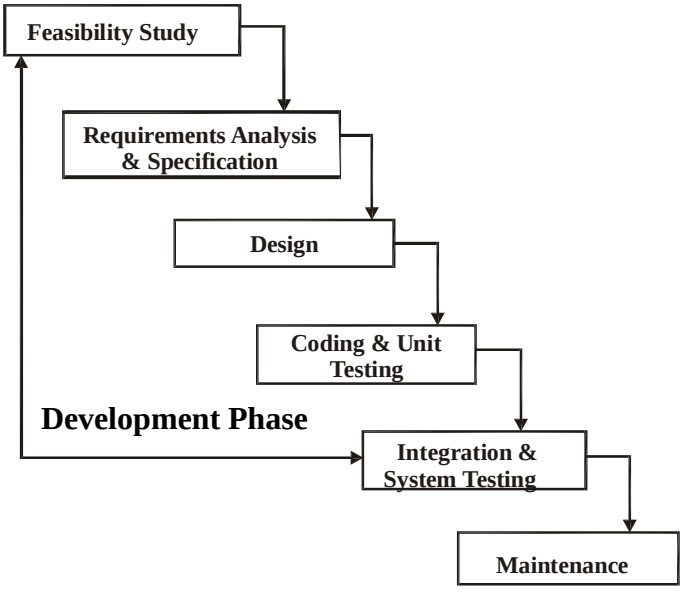


Figure: Classical Waterfall model

Now let's discuss the characteristics and working of every phase one by one.

1. Feasibility Study:

- Aim of this phase is to determine whether the system would be financially and technically feasible to develop the product.
- This is an abstract definition of the system.
- It includes the analysis of the problem definition and collection of relevant information of input, processing and output data.
- Collected data are analysed for:
  - → abstract definition Formulation of
  - → different solutions Analysis of
  - → alternative solutions
- Feasibility is evaluated from developer and customer's point of view.
- Cost/Benefit analysis is also performed at this stage.
- Three main issues are concerned with feasibility study:
  - → **Technical Feasibility** → contains hardware, software, network capability, reliability and availability.
  - → **Economical Feasibility** → contains cost/benefit analysis.
  - → **Operational Feasibility** → checks the usability. Concerned with technical performance and acceptance within the organization.

2. Requirement Analysis and Specification:

- Aim of this phase is to understand the exact requirements of the customer and to document them properly.
- It also reduces communication gap between developers and customers.
- Two different activities are performed during this phase:
  1. Requirements gathering and analysis
  2. Requirements specification
- **Requirement gathering:** The goal of this activity is to collect all relevant information from the customer regarding the product to be developed.
- **Requirements analysis:** Analyse all gathered requirements to determine exact needs of the customers/clients and hence improve the quality of the product.
- **Requirements specification:** During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document.
- The important components of SRS are → the functional requirements, the non-functional requirements and the constraints of the system.
- Output of this phase is → SRS document, which is also called Black box specification of the problem.
- This phase concentrates on "what" part, not "how".

 Requirement gathering and requirement analysis & specification collectively called **'Requirement Engineering'.**

3. Design:

- The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language.
- This phase affecting the quality of the product.
- Two main approaches are concerned with this phase:
  1. Traditional design approach
  2. Object oriented design approach

Traditional design approach:

- This approach divided into two parts: structure analysis and structure design.

Structural Analysis:

- Where the detailed structure of the problem is examined.
- Identify the processes and data flow among these processes.
- DFD is used to perform the structure analysis and to produce result.
- In structure analysis functional requirement specified in SRS are decomposed and analysis of data flow is represented diagrammatically by DFD.

Structure Design:

- During structured design, the results of structured analysis are transformed into the software design.
- Two main activities are associated with it :
  - → **Architectural design (High-level design) -** decomposing the system into modules and build relationship among them.
  - → **Detailed design (Low-level design) -** identified individual modules are design with data structure and Algorithms.

Object oriented design approach:

- In object-oriented approach, objects available in the system and relationships between them are identified.
- This approach provides lower development time and effort.
- Several tools and techniques are used in designing like:

Flow chart	DFD	Data Dictionary
Decision Table	Decision Tree	Structured English

4. Coding and Unit testing:

- It is also called the implementation phase.
- The aim of this phase is to translate the software design into source code and unit testing is done module wise.
- Each component of the design is implemented as a program module, and each unit is tested to determine the correct working of the system.
- The system is being operational (working conditions) at this phase.
- This phase affects testing and maintenance.
- Simplicity and clarity should be maintained.
- Output of this phase is → set of program modules that have been individually tested.

5. Integration and system testing:

- Once all the modules are coded and tested individually, integration of different modules is undertaken.
- The modules are integrated in a planned manner.
- This phase is carried out incrementally over a number of steps and during each integration step, the partially integrated system is tested and a set of previously planned modules are added to it.
- Finally, when all the modules have been successfully integrated and tested, system testing is carried out.
- Goal of this phase is → to ensure that the developed system works well to its requirements described in the SRS document.
- Basic function of testing is to detect errors. So, it is also called 'quality control measure'.
- Testing procedure is carried out using test data: Program test and System test. Program test is done on test data and system test is done actual data.
- Proper test cases should be selected for successful testing.
- It consists of three different kinds of testing activities.
  - → **α- testing:** It is the system testing performed by the development team.
  - → **β-testing:** It is the system testing performed by a friendly set of customers.
  - → **Acceptance testing:** It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

Output of this phase is →system test plan and test report (error report).

6. Maintenance:

- It requires maximum efforts among all the phases to develop software product.
- This phase is needed to keep system operational.
- Generally, maintenance is needed due to change in the environment or the requirement of the system.
- Maintenance involves performing following four kinds of activities:
  - → **Corrective maintenance** - Correcting errors that were not discovered during the product development phase. It is done after product development.
  - → **Perfective maintenance** - Improving and enhancing the functionalities of the system according to the customer's requirements.

    It mainly deals with implementing new or changed user requirements and improving system performance.
  - → **Adaptive maintenance** - Porting the software to work in a new environment and ensure working. It also consists of adaption of software in the changes of environment.
  - → **Preventive maintenance** - is scheduled, routine maintenance to keep equipment running as well as prevent downtime and expensive repair cost. It reduces the likelihood of failure.

In this phase, the system is reviewing for studying the performance and knowing the full capabilities of the system.

▪ **Advantages of waterfall model:**

- It is simple and easy to understand and use.
- Clearly defined stages. As each phase has well defined input and outputs.
- It helps project personnel in planning.
- Waterfall model works well for smaller projects where requirements are very well understood.

- Well understood milestones.
- Results are well documented.

▪ **Disadvantages of waterfall model:**

- High amounts of risk and uncertainty.
- It is a theoretical model, as it is very difficult to strictly follow all the phases in all types of projects.
- Not so good for complex and object-oriented projects. Also, it cannot be used for the projects where requirements are changed frequently.
- It may happen that the error may be generated at any phase and encountered in later phase. So it is not possible to go back and solve the error in this model.

▪ **Applications of waterfall model (When to use waterfall model):**

→ This model is used only when the requirements are very well known, clear and fixed.

→ When environment is stable.

→ When product definition is stable.

→ When technology is understood.

→ When the project is small.

(2) Iterative Waterfall model:

- Classical waterfall model is idealistic: It assumes that no defect is introduced during any development activity.
- But in practice defects do get introduced in almost every phase of the life cycle. Even defects may get at much later stage of the life cycle. So, solution of this problem is iterative waterfall model.
- Iterative waterfall model is by far the most widely used model. Almost every other model is derived from the waterfall model.
- The principle of detecting errors as close to its point of introduction as possible - is known as "phase containment of errors."
- Phase containment of errors can be achieved by reviewing after every milestone.

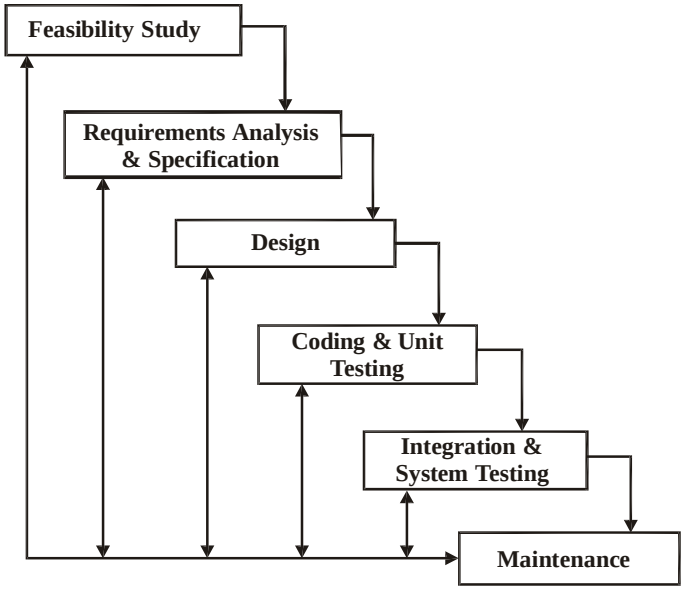


Figure: Iterative Waterfall Model

(3) Incremental Model:

- It is also referred as the successive version of waterfall model using incremental approach and evolutionary model.
- In this model, the system is broken down into several modules which can be incrementally implemented and delivered.
- First develop the core product of the system. The core product is used by customers to evaluate the system.
- The initial product skeleton is refined into increasing levels of capability : by adding new functionalities in successive versions.

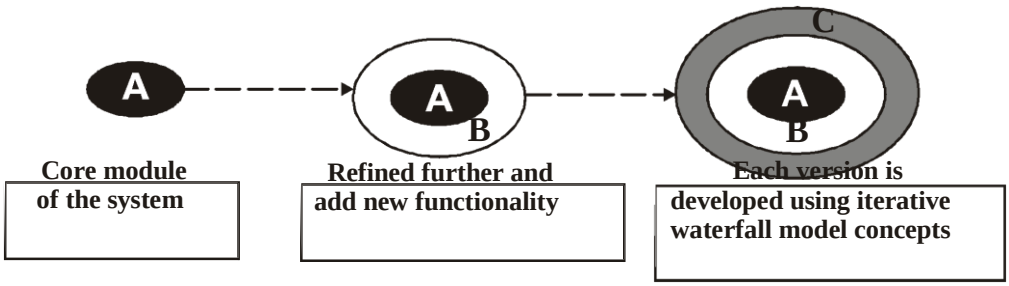


Figure: Incremental model

- From above figure, we can analyse that at the initial stage, core module/product is developed. Then by adding customer requirements or new functionalities the incremental version is built. Each version is developed using iterative waterfall model concepts.

Advantages:

→ Each successive version performing more useful work than previous versions.

→ Initial product deliver is faster.

→ The core modules get tested thoroughly, thereby reducing chance of errors in final product.

→ The model is more flexible and less costly to change the scope and requirements.

→ User gets a chance to experiment with partially developed software.

→ This model helps finding exact user requirements.

→ Feedback providing at each increment is useful for determining the better final product.

Disadvantages:

→ Sometimes it is difficult to subdivide problems into functional units.

→ Model can be used for very large problems.

→ It needs good planning and design.

→ Resulting product cost may exceed.

Applications:

→ When the problem is very large and user requirements are not well specified at initial stage.

→ When new technology is used in development.

(4) RAD model:

- Full form of RAD is - Rapid Application Development. This model was proposed by IBM in 1980.
- Rapid application development (RAD) is an incremental software development process model that emphasize on extremely short development cycle.
- It emphasizes on reuse.
- The RAD model is a "high-speed" adaptation of the linear sequential model in which rapid development is achieved by using component-based construction.
- If requirements are well understood and project scope is limited, the RAD process enables a development team to create a "fully functional system" within very short time periods (e.g., 60 to 90 days).
- Figure of this model is shown below. Many development teams are working parallel to complete the task.

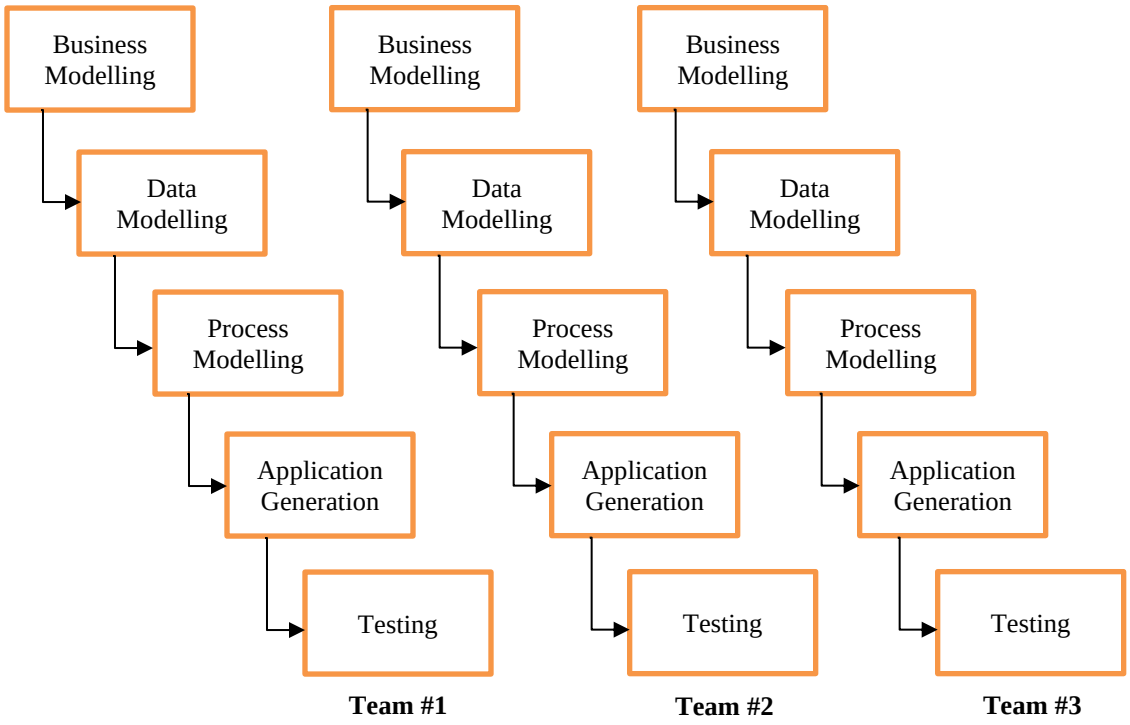


Figure: RAD Model

- **Phases of RAD model are:**

1. **Business modeling**

- The information flow among business functions is carried out in this phase. Business functions are modelled.

2. **Data modeling**

- The characteristics of objects (attributes) and their relationships are defined.

3. **Process modeling**

- The data objects defined in data modeling are transformed to implement business functions.

4. **Application generation**

- Automated tools are used to convert process models into code and the actual system. Tools like VB, VC++ and JAVA etc. are used.
- RAD works to reuse existing components or create new ones.

5. **Testing and turn over**

- As RAD uses the components that already been tested so the time for developing and testing is reduced.

Advantages:

→ Application can be developed in a quick time.

→ This model highly makes use of reusable components.

→ Reduce time for developing and testing.

→ Due to full customer involvement, customer satisfaction is improved.

Disadvantages:

→ Requirements must be cleared and well understood for this model.

→ It is not well suited where technical risk is high.

→ In it, highly skilled and expert developers are needed.

→ User involvement is necessary.

Applications:

→ The RAD model is mostly used when the system is modularized and all the requirements are well defined.

→ When a system needs to be produced in a short span of time (2-3 months).

→ RAD SDLC model should be chosen only if resources with high business knowledge are available.

→ When the technical risk is less.

(5) Prototype model:

- Prototype is a working physical system or sub system. Prototype is nothing but a 'toy implementation of a system'.
- In this model, before starting actual development, a working prototype of the system should be built first.
- A prototype is actually a partial developed product.
- A prototype usually turns out to be a very crude version of the actual system.
- Compared to the actual software, a prototype usually has:
  - o limited functional capabilities
  - o low reliability
  - o inefficient performance
- Prototype usually built using several shortcuts, and these shortcuts might be inefficient, inaccurate or dummy functions.
- Prototype model is very useful in developing GUI part of system. The prototype model is shown in below figure.

In working of the prototype model, product development starts with initial requirements gathering phase.

- Then, quick design is carried out and prototype is built.
- The developed prototype is then submitted to the customer for his evaluation.
- Based on customer feedback, the requirements are refined and prototype is modified.
- This cycle of obtaining customer feedback and modifying the prototype continues till the customers approve the prototype.
- Then the actual system is developed using the different phases of iterative waterfall model.
- There are two types of prototype model.

I. Exploratory development prototype model: in which the development work is done with the users to explore their requirements and deliver a final system.

II. Throw away prototype model: in which a small part of the system is developed and given to the customer to try out and evaluate. Then feedback is collected from customer and accordingly main system is modified. The prototype is then discarded.

Advantages:

→ A partial product is built at initial stage, so customers can get a chance to have a look of the product.
→ New requirements can be accommodated easily.
→ Missing functionalities identified quickly.
→ It provides better flexibility in design and development.
→ As the partial product is evaluated by the end users, more chance of user satisfaction.
→ Quick user feedback is available for better solution.

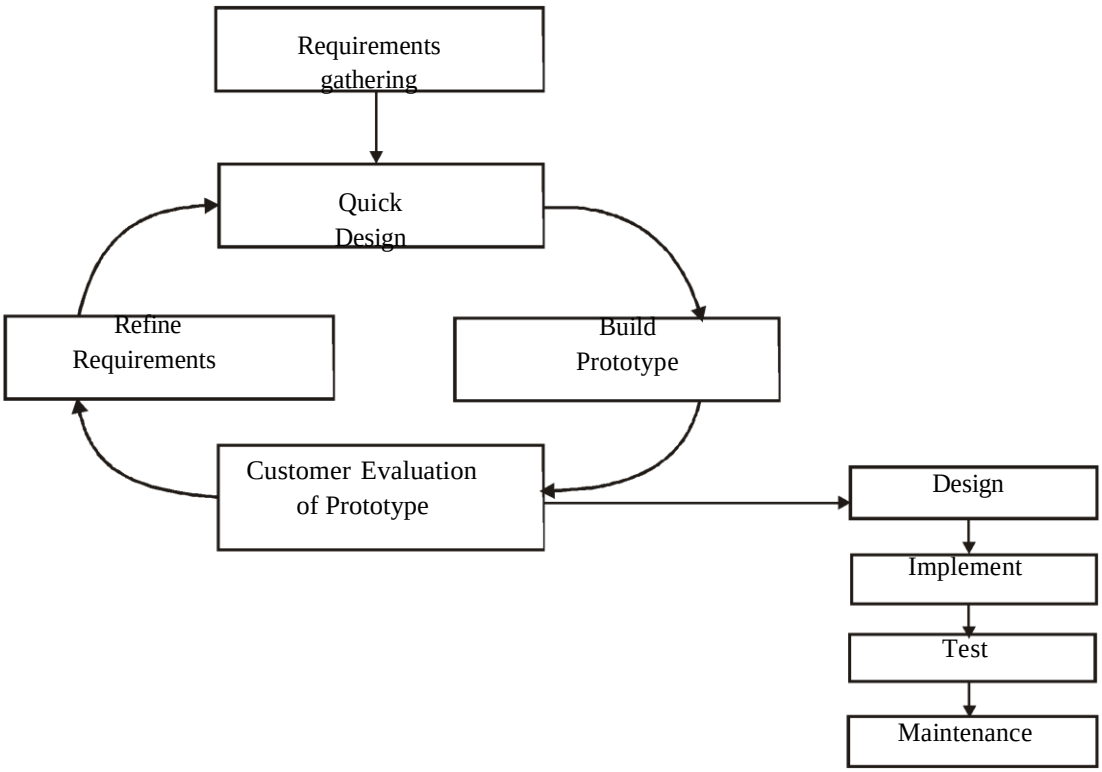


Figure: The Prototype Model

Disadvantages:

- → The code for prototype model is usually thrown away. So, wasting of time is there.
- → The construction cost of developing the prototype is very high.
- → If end user is not satisfied with the initial prototype, he may lose interest in the final product.
- → This model requires extensive participation and involvement of the customers that is not possible every time.

Applications:

- → This model used when desired system needs to have a lot of interactions with end users.
- → This type of model generally used in GUI (Graphical User Interface) type of development.

(6) Spiral model:

- Spiral model is proposed by Boehm in 1986.
- In application development, spiral model uses fourth generation languages (4GL) and developments tools.
- The diagrammatic representation of this model appears like a spiral with many loops.

Figure of spiral model is shown below.

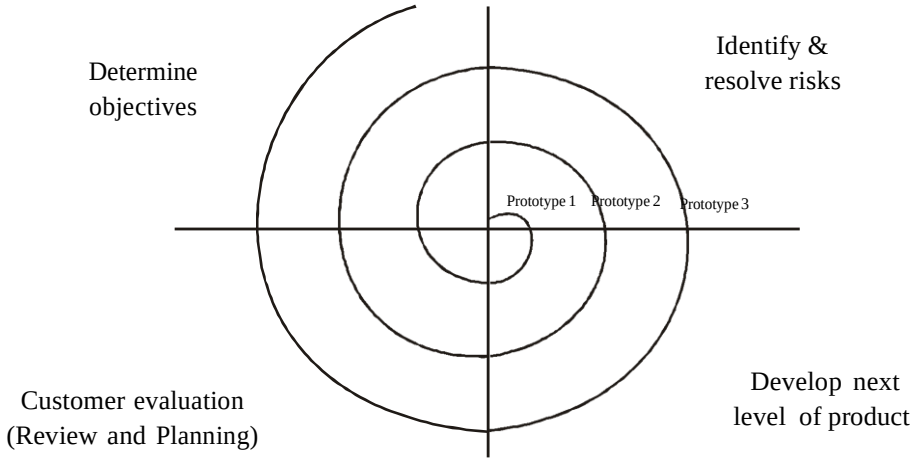


Figure: Spiral Model

Each loop of the spiral represents a phase of the software process:

- The innermost loop might be concerned with system feasibility,
- The next loop with system requirements definition,
- The next one with system design, and so on
- Number of phases is not fixed in this model.
- This model is more flexible compared with other models.
- Each loop in the spiral is split into four sectors (quadrants).

1st Quadrant- Determine objectives:

- Identifying the objectives, identifying their relationships and find the possible alternative solutions. Objectives like: performance, functionality, hardware software interface etc.
- Also examine the risks associated with these objectives.

2nd Quadrant- Identify and resolve risks:

- In this phase, detailed analysis is carried out of each identified risk
- Alternate solutions are evaluated and risks are reduces at this quadrant.

3rd Quadrant - Develop next level product:

- Develop and validate the next level of the product after resolving the identified risks.
- Activities like design, code development, code inspection, testing and packaging are performed.
- Resolution of critical operations and technical issues of next level product are performed.

4th Quadrant - Review and Planning (Customer evaluation):

- In this part, review the results achieved so far.
- Plan the next iteration around the spiral. Different plans like development plan, test plan, installation plan are performed.
- With each iteration around the spiral; progressively more complete version of the software gets built.
- In spiral model at any point, Radius represents cost and Angular dimension represents progress of the current phase.
- At the end, all risks are resolved and software is ready to use.

Advantages:

- It is more flexible, as we can easily deal with changes.
- Due to user involvement, user satisfaction is improved.
- It provides cohesion between different stages.
- Risks are analyzed and resolved so final product will be more reliable.
- New idea and additional functionalities can be easily added at later stage.

Disadvantages:

- It is applicable for large problem only.
- It can be more costly to use.
- It is more complex to understand.
- More number of documents are needed as a greater number of spirals.
- Risk analysis phase requires much higher expertise.

Applications (when to use spiral model):

- Used when medium to high-risk projects.
- When users are unsure for their needs.
- When requirements are complex.

Spiral model can be viewed as a Meta model.

- Spiral model subsumes almost all the life cycle models.
- Single loop of spiral represents Waterfall Model.
- Spiral model uses a prototyping approach by first building the prototype before developing actual product.
- The iterations along the spiral model can be considered as supporting the Evolutionary Model.
- The spiral model retains the step-wise approach of the waterfall model.

- **Comparison of different Life cycle models :**

- **The Classical waterfall model**

→ Can be considered as a basic model as all other models are derived from this model.

→ But, it is not used in practical development as no mechanism to handle errors committed during the phase.

- **The Iterative waterfall model**

→ Most widely used model.

→ But, suitable only for well-understood problems.

- **The Prototype model**

→ Suitable for projects in which user requirements and technical aspects are not well understood.

→ Especially popular for user interface part of the project (GUI based projects).

- **The Evolutionary model**

→ It is suitable for large problems: as it can be decomposed into a set of modules that can be incrementally implemented.

→ Also used for object-oriented development projects.

→ But can be used only if the incremental delivery of the system is acceptable by the customer.

- **The Spiral model**

→ Used to develop the projects those have several kinds of risks.

→ But, it is much complex than other models.

- **Program versus software product.**

Parameter	Program	Software product
Size	It is usually small in size.	It is large in size.
Developer	It has single developer.	Here, team of developers.
User	Author himself is sole user.	Large number of users.
Interface	It lacks proper user interface.	Here, well designed interface.
Documentation	It lacks proper documentation.	Here, well documented and user manual prepared.
Functionality	It provides limited functionality and less features.	It provides more functionality.
Development	No need of systematic, organized and planned development.	Require proper systematic, organized and planned development.

Unit-2

Requirement Analysis & Specification

❖ REQUIREMENT GATHERING & ANALYSIS

- Requirements of a customer play a key role in developing any software product.
- The task of gathering requirements and analyzing them is performed by a System analyst.
- Collecting all the information from the customer and then analyze the collected information to remove all ambiguities and inconsistencies from customer perception.
- Mainly two activities are concerned with this task.

Requirement gathering	Requirement analysis

Requirement gathering:

- It is usually the first part of any software product.
- This is the base for the whole development effort.
- The goal of the requirement gathering activity is → to collect all related information from the customer regarding the product to be developed.
- This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.
- In this phase, meeting with customers, analyzing market demand and features of the product are mainly focused.
- So, activity of market research (for competitive analysis) is done.
- It involves interviewing the end-users and studying the existing documents to collect all possible information.

Requirement gathering activities are:

- o Studying the existing documents
- o Interview with end users or customers
- o Task analysis
- o Scenario analysis
- o Form analysis
- o Brainstorming
- o Questionnaires
- o Group discussion

Requirement analysis:

- The goal of the requirement analysis activity is → to clearly understand the exact requirements of the customers.
- **IEEE** defines requirement analysis as (1) the process of studying user needs and (2) The process of studying and refining system hardware or software requirements.
- Requirement analysis helps to understand, interpret, classify, and organize the software requirements in order to assess the feasibility, completeness, and consistency of the requirements.

Requirement analysis involves:

- *Eliciting requirements*: requirements are eliciting by communicating with customers and find their exact need.
- *Analyzing requirements*: requirements are then analyzed to make it complete, clear and unambiguous.
- *Requirements recording or storing*: all the requirements are recorded in form of use cases, process specifications, natural language documents etc.

System analyst should solve some of the following questions:

- What is the problem?
- What are the inputs and outputs?
- What is important to solve?
- What are the complexities?
- What are the solutions?

- Change in the environment or technical aspects may affect the requirement analysis process.
- System analyst identifies and resolves various requirements problems.
- For that, analyst has to identify and eliminate the problems of anomalies, inconsistencies and incompleteness. **Anomaly** is the ambiguity in the requirement, **Inconsistency** contradicts the requirements, and **Incompleteness** may overlook some requirements.
- Analyst detects above problems by discussing with end-users.
- Requirement analysis is necessary to develop the system that meets all the requirements of the end-user.
- Finally, make sure that requirements should be specific, measurable, timely, achievable and realistic.
- Output of this activity is → **SRS** (Software Requirements Specification).

- **Software requirements are the description of services which software will provide to end user.**
- **Requirement gathering and requirement analysis & specification collectively called 'Requirement Engineering'.**

❖ SOFTWARE REQUIREMENT SPECIFICATION (SRS)

- SRS is the output of requirement gathering and analysis activity.
- SRS is a document created by system analyst after the requirements are collected from various sources.
- SRS is a detailed description of the software that is to be developed. It describes the complete behavior the system.
- SRS describes 'what' the proposed system should do without describing 'how' the software will do (what part, not how).
- It is working as a reference document to the developer.
- It provides guideline for project development, so minimizes the time and efforts for software development.
- SRS is actually a contract between developer and end user. That helps to dissolve the disagreement.

- The SRS translates the ideas of the customers (input) into the formal documents (output).
- The SRS document is known as black-box specification, because:
  - In SRS, internal details of the system are not known (as SRS doesn't specify how the system will work).
  - Only its visible external (i.e., input/output) behavior is documented.
- SRS documents serves as contract between customer and developer, so it should be carefully written. (Sometimes SRS is also written by the customers also).
- The organization of SRS is done by the system analyst.

▪ **Benefits of SRS (Features of SRS):**

- SRS provides foundation for design work. Because it works as an input to the design phase.
- It enhances communication between customer and developer because user requirements are expressed in natural language.
- Developers can get the idea what exactly the customer wants.
- It enables project planning and helps in verification and validation process.
- Format of forms and rough screen prints can also be represented in SRS.
- High quality SRS reduces the development cost and time efforts.
- As it is working as an agreement between user and developer, we can get the partial satisfaction of the end user for the final product.
- SRS is also useful during the maintenance phase.

▪ **Contents of the SRS document:**

- An SRS should clearly document the following three things:

(i) Functional requirements of the system

- Functional requirements are those which are related to the technical functionality of the system.
- These are the services which the end users expect from the final product. And these are the services which a system provides to the end users.
- It clearly describes each of the function that the system needs to perform along with the input and output data set.

(ii) Non-functional requirements of the system

- The non-functional requirements describe the characteristics of the system that can't be expressed functionally. For example, portability, maintainability, reliability, usability, security, performance etc.
- Non-functional requirements are requirements that specify criteria that can be used to judge the operation of a system in particular conditions, rather than specific behaviors.
- Sometimes these requirements are also called quality attributes.

(iii) Constraints (restrictions) on the system

- That describes what the system should do or should not do. These are some general suggestions regarding development.
- A constraint can be classified as:
  - Performance constraint
  - Operating system constraint

- Economic constraint
- Life cycle constraint
- Interface constraint

- **Characteristics of a good**

SRS: Concise

SRS should contain brief and concise information regarding the project; no more detailed description of the system should be there.

Complete

It should be complete regarding the project, so that can be completely understood by the analyst and developers as well as customers.

Consistent

An SRS should be consistent through the project development. Requirements may not be conflict at the later stage.

Conceptual integrity

SRS should clearly provide the concepts of the system, so that can be read easily.

Structured

SRS should be well structured to understand and to implement.

Black box view

SRS should have black box view means; there should not be much detailing of the project in it (only describe what part, not how).

Verifiable

It should be verifiable by the clients or the customers for whom the project is being made.

Adaptable

It should be adaptable in both sides from the clients as well as from the developers.

Maintainable

SRS should be maintainable so in future changes can be made easily.

Portable

It should be portable as if we can use the contents of it for the same types of developments.

Unambiguous

There should not be any alternates of SRS that creates ambiguity.

Traceable

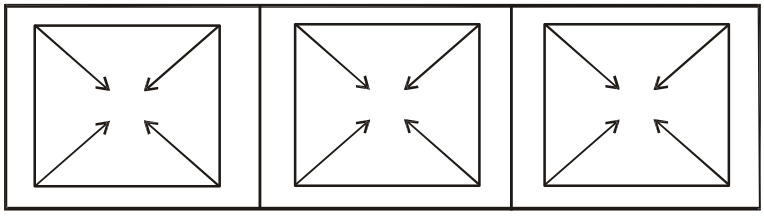
Each of the requirements should be clear and refer to the future development.

❖ COHESION AND COUPLING

- Modularity is clearly a desirable property of any software development.
- In software development, modularity is →decomposition of a program into smaller programs (or modules).
- A system is considered modular if it consists of multiple modules so that each module can be implemented separately and debugged separately.
- Modular system provides advantages like:
  - Easy to understand the system.
  - System maintenance is easy.
  - Provide reusability.

- Modularity is successful because developers use prewritten code, which saves resources. Overall, modularity provides greater software development manageability.
- Cohesion and coupling are two modularization criteria that are often used together.
- Most researchers and developers are agreed that for good software design neat decomposition is highly needed, and the primary characteristic of neat decomposition is 'high cohesion and low coupling'.

- **Cohesion:**
  - Cohesion is → a measure of functional strength of a module.
  - Cohesion keeps the internal modules together, and represents the functional strength.
  - Cohesion of a module represents how tightly bound the internal elements of a module are to one another.



Cohesion = strengths of relations within modules

Classification of cohesion:

Coincidental	Logical	Temporal	Procedural	Communicational	Sequential	Functional

Worst (Low) → Best (High)

Coincidental cohesion

- It is the lowest cohesion. Coincidental cohesion occurs when there are no meaningful relationships between the elements.
- A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely.
- It is also called random or unplanned cohesion.

Logical cohesion

- A module is said to be logically cohesive if there is some logical relationships between the elements of module, and the elements perform functions that fall into same logical class.
- For example: the tasks of error handling, input and output of data.

Temporal cohesion

- Temporal cohesion is same as logical cohesion except that the elements are also related in time and they are executed together.
- A module is in temporal cohesion when a module contains functions that must be executed in the same time span.
- Example: modules that perform activities like initialization, cleanup, and start-up, shut down are usually having temporal cohesion.

Procedural cohesion

- A module has procedural cohesion when it contains elements that belong to common procedural unit.
- A module is said to have procedural cohesion, if the set of the modules are all part of a procedure (algorithm) in which certain sequence of steps are carried out to achieve an objective.

- Example: the algorithm for decoding a message

Communicational cohesion

- A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, for example the set of functions defined on an array or a stack.
- These modules may perform more than one function together.

Sequential cohesion

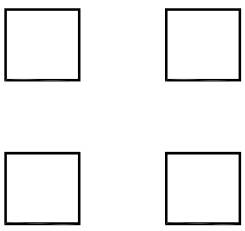
- When the output of one element in a module forms the input to another, we get sequential cohesion.
- Sequential cohesion does not provide any guideline how to combine these elements into modules.
- For example, in a TPS (transaction processing system), the get-input, validate-input, sort-input functions are grouped into one module.

Functional cohesion

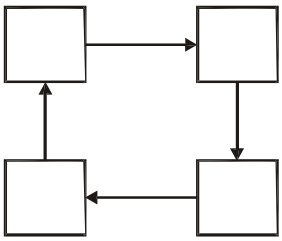
- Functional cohesion is the strongest cohesion.
- In it, all the elements of the module are related to perform a single task.
- All elements are achieving a single goal of a module.
- Functions like: compute square root and sort the array are examples of these modules.

▪ **Coupling:**

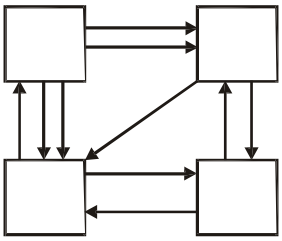
- Coupling between two modules is → a measure of the degree of interdependence or interaction between these two modules.
- Coupling refers to the number of connections between 'calling' and a 'called' module. There must be at least one connection between them.
- It refers to the strengths of relationship between modules in a system. It indicates how closely two modules interact and how they are interdependent.
- As modules become more interdependent, the coupling increases. And loose coupling minimize interdependency that is better for any system development.
- If two modules interchange large amount of data, then they are highly interdependent or we can say they are highly coupled.
- High coupling between modules makes the system difficult to understand and increase the development efforts. So low (OR loose) coupling is the best.



No coupling



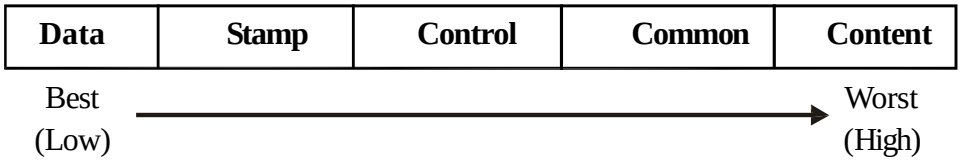
Loose coupling



High coupling

Classification of coupling:

Five different types of coupling can occur between two modules.



Data coupling

- Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between these two.
- For example → an int, a char, a float etc.
- It is lowest coupling and best for the software development.

Stamp coupling

- Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

Control coupling

- Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another module.
- An example of control coupling is a flag set in one module and tested in another module.

Common coupling

- Two modules are common coupled, if they share data through some global data items. It means two or more modules are communicating using common data.

Content coupling

- It is the highest coupling and creates more problems in software development.
- Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module.
- It is also known as 'pathological coupling'.

▪ **Functional independence:**

- A module having high cohesion and low coupling is said to be functionally independent of other modules.
- So, that a cohesive module performs a single task or function.
- A functionally independent module has minimal interaction with other modules.
- For good software design neat decomposition is highly needed, and the primary characteristic of neat decomposition is **'high cohesion and low coupling'.**

> **Intra dependency (Cohesion) between modules should be high and inter dependency (Coupling) should be low.**

Need of functional independence:

- Functional independence is a good key to any software design process due to following reasons :

1. Error isolation:

- It reduces error propagation.

- The reason behind this is →if a module is functionally independent, its degree of interaction with the other modules is less.
- So, the error of one module can't affect another module.

2. Scope of reuse:

- Reuse of a module becomes possible. Because each module does some well-defined and precise function, and the interaction of the module with the other modules is simple and minimal.
- Therefore, a cohesive module can be easily taken out and reused in a different program.

3. Understandability:

- Complexity of the design is reduced, because different modules can be understood in isolation as modules are more or less independent of each other.

Difference between functional and non-functional requirements:

Functional requirements	Non-functional requirements
These describe what the system should do.	These describe how the system should behave.
These describe features, functionality and usage of the system.	They describe various quality factors, attributes which affect the system's functionality.
Describe the actions with which the work is concerned.	Describe the experience of the user while doing the work.
Characterized by verbs.	Characterized by adjectives.
Ex: business requirements, SRS etc.	Ex: portability, quality, reliability, robustness, efficiency etc.

Difference between Cohesion & Coupling:

Cohesion	Coupling
Cohesion is the indication of the relationship within module.	Coupling is the indication of the relationships between modules.
Cohesion shows the module's relative functional strength.	Coupling shows the relative interdependence among the modules.
Cohesion is a degree (quality) to which a component / module focuses on the single thing.	Coupling is a degree to which a component / module is connected to the other modules.
While designing you should go for high cohesion. i.e. a cohesive component/ module focus on a single task with little interaction with other modules of the system.	While designing you should go for low coupling i.e., dependency between modules should be less.
Cohesion is the kind of natural extension of data hiding for example, class having all members visible with a package having default visibility.	Making private fields, private methods and nonpublic classes provides loose coupling.
Cohesion is Intra - Module Concept.	Coupling is Inter -Module Concept.

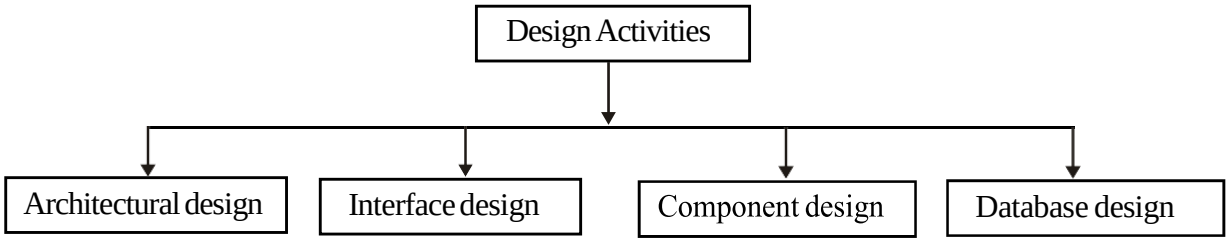
Unit-3
Software Design with UML

❖ DESIGN PROCESS

- The design process is a sequence of steps to describe all aspects of the software.
- Design process specifies how aspect of the system (means how the system will be implemented and how it will work).
- The purpose of the design process is → to plan a solution of problems specified in SRS.
- Design process includes user interface design, input output design, data design, process and program design and technical specification etc.
- It transforming the customer requirements (described in SRS) into appropriate form that is suitable to implement using any of programming languages.
- Output of design process is →design documents.

➢ Classification of design activities:

- Design activities are depending on the type of the software being developed.
- Design activities can be classified like:



Design Activities

1. **Architectural design:**
   - Where you can identify the overall structure of the system, sub-systems, modules and their relationships.
   - It defines the framework of the computer based system.
   - It can be derived from DFD (data flow diagram).
2. **Interface design:**
   - Where you can define the interface between system components.
   - It describes how system communicates with itself and with the user also.
   - It can be derived from DFD and State transition diagram.
3. **Component design:**
   - That defines each system component and show how they operate.
   - It can be derived from State transition diagram.

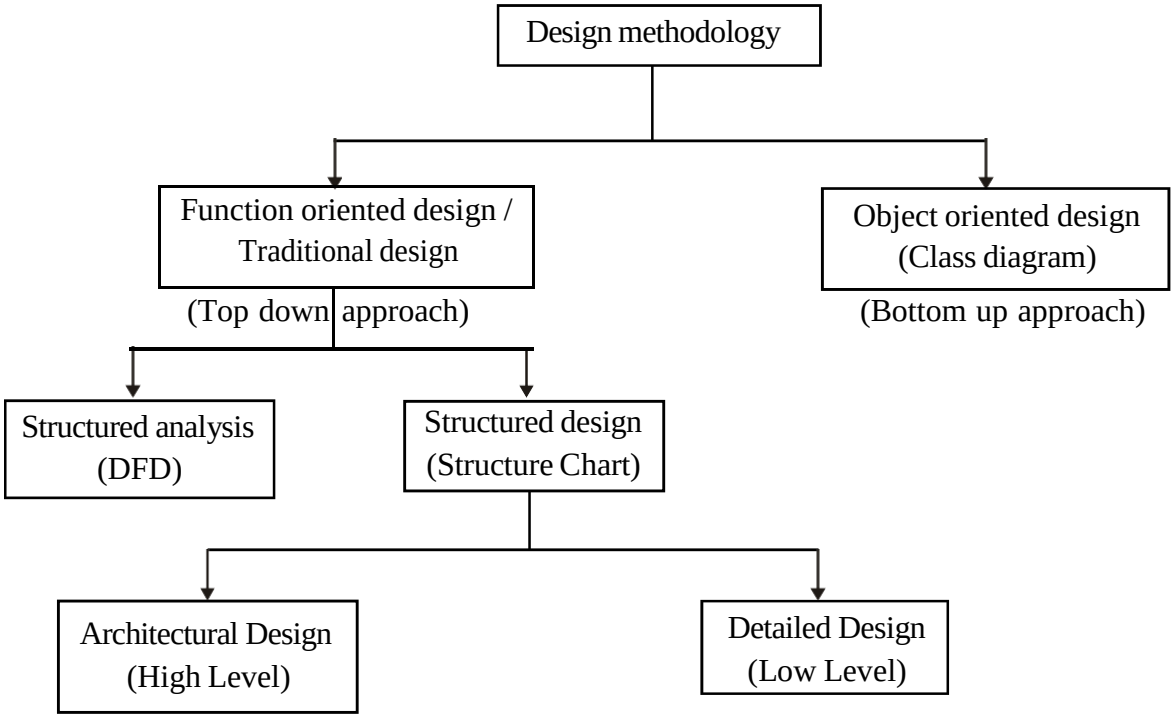
4. Database design:

- Where you can define the system data structure and show how they are represented in a database.
- Existing database can be reused or a new database to be created.
- The data objects and their relationships are defined in ERD (entity relationship diagram) and detailed content described in data dictionary (DD).
- It can be derived from ERD and DD.

➢ Classification of design methodologies:

- Design methodologies are followed in software development from beginning up to the completion of the product.
- Design methodologies use to provide guidelines for the design activity.
- The nature of the design methodologies are dependent on the following factors:
  - → The software development environment.
  - → The type of the system being developed.
  - → User requirements.
  - → Qualification and training of the development team.
  - → Available software and hardware resources.
- There are large numbers of software design methodologies. Different methodologies are used to solve different type of problems.

Classification of design methodologies is shown in the figure.



Design Methodologies

- There are fundamentally two different approaches :

Function oriented design	**Object oriented design**

1. Function oriented design:

- In which the set of functions are described.
- Starting at this high level view of the system, each function is successively refined into more detailed functions or sub functions (top down approach).
- Each of these sub-functions may be split into more detailed sub functions and so on.
- Data in the system is centralized and shared among different functions.
- Function oriented design further classified into → Structure analysis and Structure design.

➢ Structure analysis

- It is used to transform a textual description into graphical form.
- It examines the detail structure of the system.
- It identifies the processes and data flow among these processes.
- In structure analysis - functional requirements specified in SRS are decomposed and analysis of data flow is represented here.
- Structure analysis is graphically represented using data flow diagram (DFD).

➢ Structure design

- During structured design, the results of structured analysis are transformed into the software design.
- All the functions identified during analysis are mapped to module structure and that is useful for implementation.
- The aim of the structure design is → to transform the result of the structure analysis (DFD) into a structure chart. So structure analysis is graphically represented using structure chart.
- Two main activities are performed during structure analysis :
  - **(i) Architectural design (High level design):** decomposing the system into modules and build relationship among them.
  - **(ii) Detailed design (Low level design):** individual modules are design with data structure and algorithms.
- Problem with the structure design is that → if a change is made in one part of the program will require search through of entire program.

2. Object oriented design:

- In this design the system is viewed as a collection of objects (entities).
- Objects available in the systems are identified and their relationships are built during this approach. And the whole system is built (bottom up approach).
- Each object has its own internal data and similar objects constitute a class. So each object is a member of a class.
- Class diagram is used to represent the object oriented design. UML modeling and use case are also used in object oriented design.
- Data in the system is not centralized and shared but is distributed among the objects in the system.
- Three main concepts: Encapsulation, Inheritance and Polymorphism greatly enhance the ability of developers' to more easily manage the software product.
- **Encapsulation** combining data and functions into a single entity, **inheritance** provide reusability and by **Polymorphism** same context can be used for different purposes.

➢ Advantages of object oriented design:

- Reduce maintenance.
- Provide code reusability, reliability, modeling, reliability and flexibility.
- Provide robustness to the system.
- It provides roadmap for reusing objects in new software.
- The object oriented design process is consistent from analysis, through design to coding.

❖ DATA MODELING CONCEPTS

- A data model is a conceptual relationship of data structure (tables) required for a database. And it is concerned with the structure rather than rules.
- To avoid the redundancy of data in OLTP (online transaction process) database, there is a need to create data model.
- Data model provides abstract and conceptual representation of data.
- Data modeling or ER diagram gives the concepts of objects, attributes and relationship between objects.
- ER diagram is a structured analysis technique. And also describes logical data design that can converted easily into table structure.
- ERD is a snapshot of data structure.
- ER diagram can be used to model the data in the system and how the data items relate to each other but do not cover how the data is to be processed.
- ER diagram enables a software engineer to identify data objects and their relationships using graphical notations.
- ERD is a detailed logical representation of any system. It has three main elements → data object (entity), attributes and their relationships.

➢ Data object (Entity set):

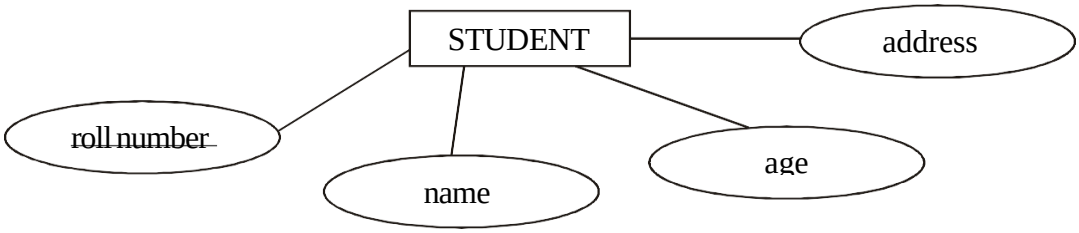
- A data object is a real world entity or thing.
- Data object is a fundamental composite information system.
- An entity → represents a thing that has meaning and about which you want to store or record data.
- It can be external entity, a thing, an organization, a place or an event.
- For example: for a college → department, students, head of the department may be entities.
- It has number of properties or attributes. Each object has its own attributes.
- Entities are represented using rectangle box and preferably entity name is written in capital letters.

Entity set for a college

➢ Data Attributes :

- An attribute is a property or characteristic of an entity.
- Attributes provide meaning to the objects.
- Attributes must be defined as an identifier, and that become key to find instance of object.

- For example: attributes for entity student is roll number, name, age, address etc.
- Attributes represented using oval.

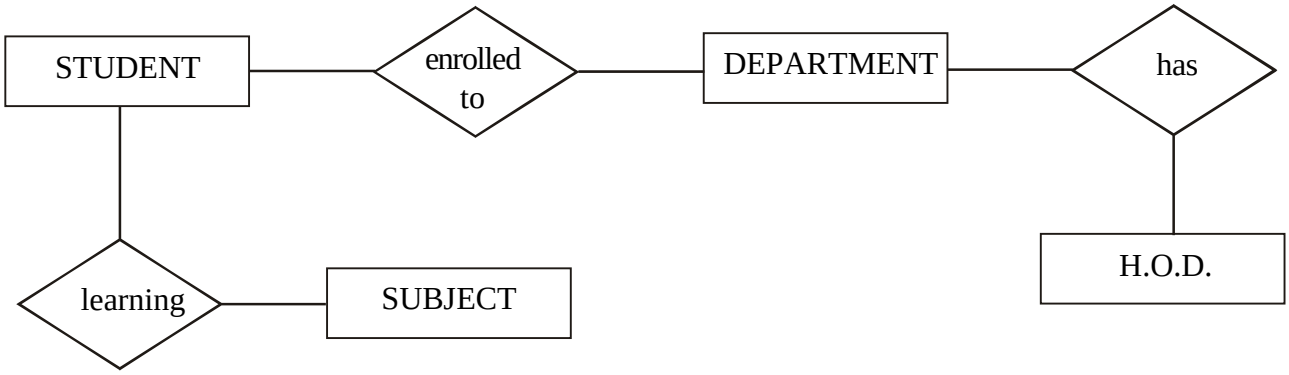


Attributes of entity student

(Note: attribute 'roll number' is underlined because it is key attribute)

➢ Relationship:

- Entities are connected to each other via relations. Generally relationship is binary because there are two entities are related to each other.
- Relationship illustrates how two entities share information in the database structure.
- Relationship of objects is bidirectional, so they can be read in either side.
- Relationship is represented using diamond shape symbol with joined relationship name.
- Below figure gives the understanding of relationship between student, department, head of the department and subject.



Relationship of entities

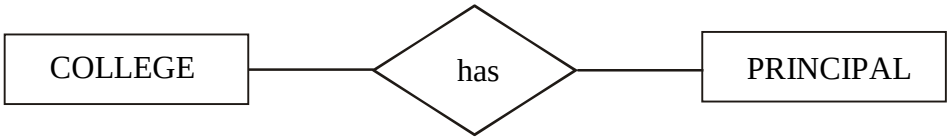
➢ Cardinality :

- The elements of data modeling - data objects, attributes, and relationships, provide the basis for understanding the information domain of a problem. However, additional information related to these basic elements must also be understood.
- Object X has relationship to object Y does not provide enough information for software engineering purposes. We must understand how many occurrences of object X are related to how many occurrences of object Y. This leads to a data modeling concept called cardinality.
- The concept of cardinality defines the maximum number of objects that can participate in a relationship. That means number of occurrences of one [object] that can be related to the number of occurrences of another [object].
- Cardinality is usually expressed as simply 'one' or 'many.' For example a father can have one or more children but a child can have only one father.

- Maximum cardinality means maximum number of instance attached in relationship and minimum in vise versa.
- Different cardinalities are explained below.

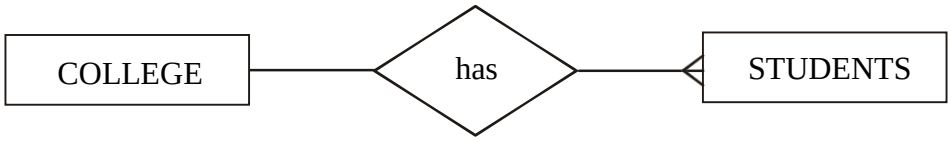
➢ One to One (1 : 1)

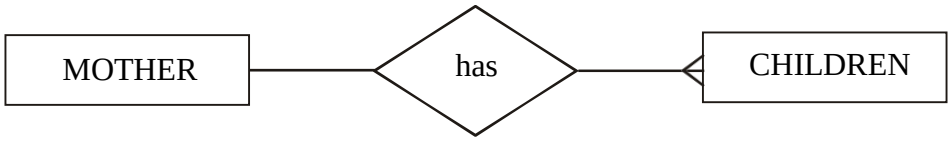
- An instance (occurrence) of entity (object) A can relate to one only instance (occurrence) of B and instance of B can relate to only one instance of A.
- For example one college has only one principal and one principal can take charge of one college only.



➢ One to Many (1 : M)

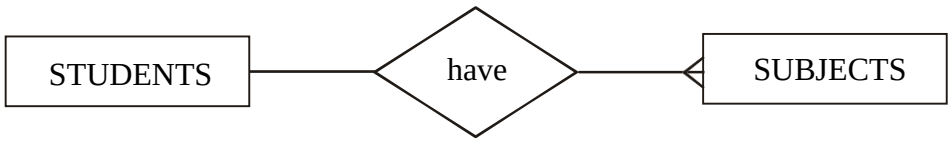
- One instance of entity A can relate to one or many instances of B, but an instance of B can relate to only one instance of entity A.
- A mother can have many children but a child can have only one mother. In another example, one college can have many students but a student can be enrolled to one college.

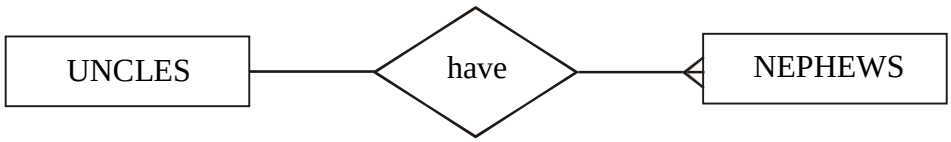




➢ Many to Many (M : M)

- One or more instances of entity A can relate to one more instances of entity B. and vice versa.
- For example an uncle can have many nephews while a nephew can have many uncles. In another example, many students can registered with one or more subjects as well one subject can be registered by one or more students.

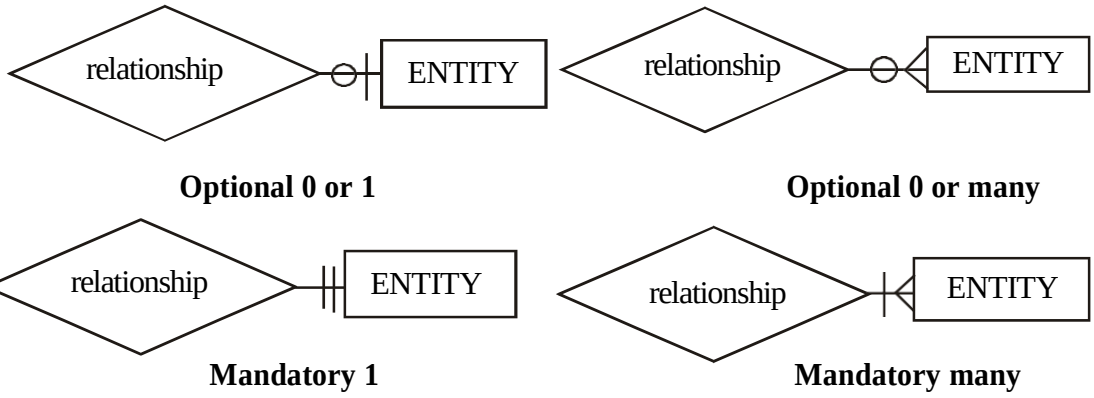




➢ Modality:

- Cardinality does not, however, provide an indication of whether or not a particular data object must participate in the relationship. To specify this information, the data model adds modality concept to the object/relationship pair.
- Modality is the form of cardinality.

- Modality means a classification of relationships on the basis of whether they claim necessity, possibility or impossibility.
- The modality of relationship is 0 if there is no explicit need for the relationship or the relationship is 0 or it is optional.
- The modality is 1 if an occurrence of relationship is mandatory.
- The notations for modality are explained below.



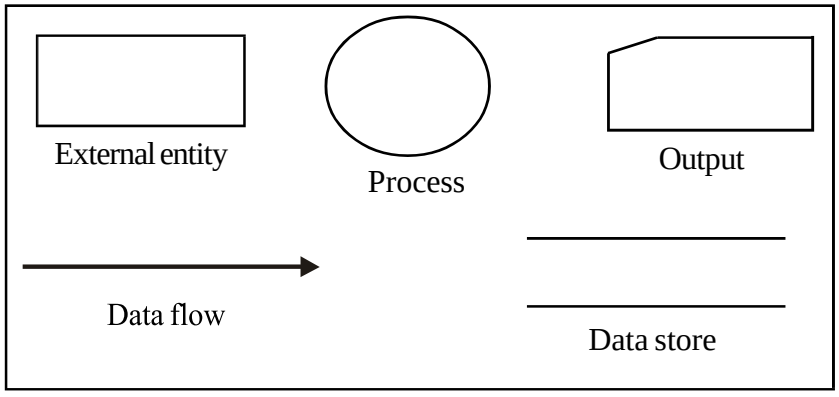
(**Note:** There are many different notations available for cardinality and modality, like Chen's notations, Crow's foot notation etc. So it may be possible that you may see different notations in various books or websites.)

❖ DATA FLOW DIAGRAMS

- DFD (Data Flow Diagram) is also known as bubble chart or data flow graph.
- DFDs are very useful in understanding the system and can be effectively used during analysis. It shows flow of data through a system visually.
- The DFD is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among these functions.
- It views a system as a function that transforms the inputs into desired outputs.
- Each function is considered as a process that consumes some input data and produces some output data.
- The system is represented in terms of the input data to the system, various processing carried out on these data, and the output data generated by the system.
- Functional model can be represented using DFD.

➢ Primitive symbols used in construction of DFD model:

- The DFD model uses a very limited number of primitive symbols.



DFD symbols

1. **Process (function)**
   - Process or function is represented by circle or bubble.
   - Circles are annotated with names of the corresponding functions.
   - A process shows the part of the system that transforms inputs into outputs.
     - The process is named using a single word that describes what the system does functionally. Generally processis named using 'verb'.
2. **External entity**
   - Entity is represented by a rectangle. Entities are external to the system which interacts by inputting data to the system or by consuming data produced by the system.
   - It can also define source (originator) or destination (terminator) of the system.
3. **Data flow**
   - Data flow is represented by an arc or by an arrow.
   - It used to describe the movement of the data.
   - It represents the data flow occurring between two processes, or between an external entity and a process. It passes data from one part of the system to another part.
   - Data flow arrows usually annotated with the corresponding data names. Generally data flow named using 'noun'.
4. **Data store**
   - Data store is represented by two parallel lines.
   - It is generally a logical file or database.
   - It can be either a data structure or a physical file on the disk.
5. **Output**
   - Output is used when a hardcopy is produced.
   - It is graphically represented by a rectangle cut either a side.

➢ **Developing DFD model of the system:**

- DFD graphically shows the transformation of the data input to the system to the final result through a hierarchy of levels.
- DFD starts with the most abstract level of the system (lowest level) and at each higher level, more details are introduced.
- To develop higher level DFDs, processes are decomposed into their sub functions.
- The abstract representation of the problem is also called context diagram.

➢ **Context diagram (Level 0 DFD)**

- The context diagram is top level diagram; it is the most abstract data flow representation of a system.
- It is an overall, simplified view of target system.
- It only contains one process node that generalizes the function of entire system with external entities. (Itrepresents the entire system as a single bubble.)
- Data input and output are represented using incoming and outgoing arrows. These arrows should be annotated with the corresponding data items (usually a noun).

➢ Level 1 diagram

- To develop the level 1 DFD, we have to examine the high-level functional requirements.
- It is recommended that 3 to 7 functional requirements can be directly represented as bubbles in 1st level.

➢ Further Decomposition (Level 2 and above)

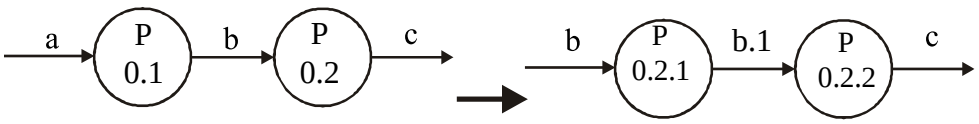
- The bubbles are decomposed into sub-functions at the successive levels of the DFD.
- Decomposition of a bubble is also known as factoring or exploding a bubble.
- Each bubble at any level of DFD is usually decomposed between 3 to 7 bubbles in its higher level.
- Successive levels give more detailed description about the system.
- It's not a rule that particular number of levels are needed for the system, but decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm.

➢ Numbering the bubbles

- It is necessary to number the different bubbles occurring in the DFD.
- These numbers help in uniquely identifying any bubble in the DFD by its bubble number.
- The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD.
- Bubbles at level 1 are numbered, 0.1, 0.2, 0.3, etc.

➢ Balancing DFD

- The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is known as balancing a DFD.



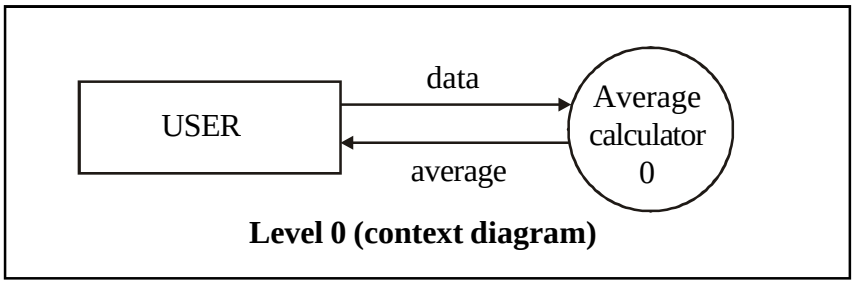
Data flow in level - 1 **Data flow in level - 2**

➢ Some care should be taken while constructing DFDs (Guidelines when drawing DFDs) :

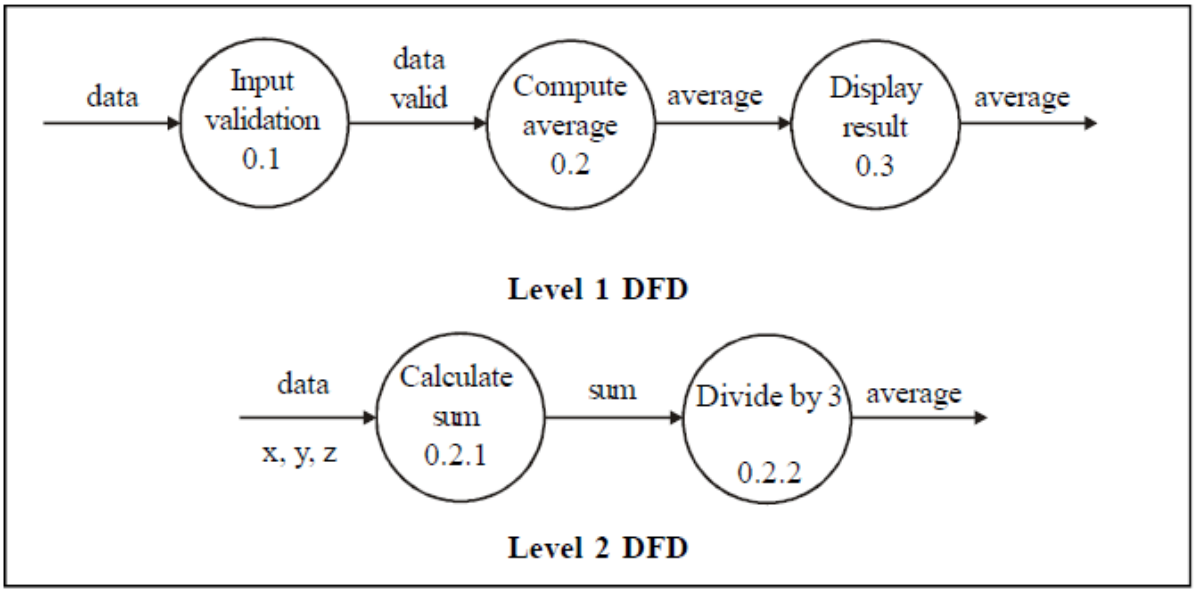
- A process must have at least one input and one output data flow.
- No control information (IF-THEN-ELSE) should be provided in DFD.
- A data store must always be connected with a process. A data store cannot be connected to another data store or to an external entity.
- Data flows must be named.
- Data flows from entities must flow into processes, and data flows to entities must come from processes.
- There should not be detailed description of process in context diagram.
- Name of the data flow should be noun and name of process should be verb.
- Each low level DFD must be balanced to its higher level DFD (input and output of the process must be matched in next level).
- Data that travel together should be one data flow.
- No need to draw more than one bubble in context diagram.
- Generally all external entities interacting with the system should be represented only in the context diagram.

- Be careful with number of bubbles in particular level DFD, as too less or too many bubbles in DFD are oversight.
- All the functionalities of the system specified in SRS must be captured by the DFD model.

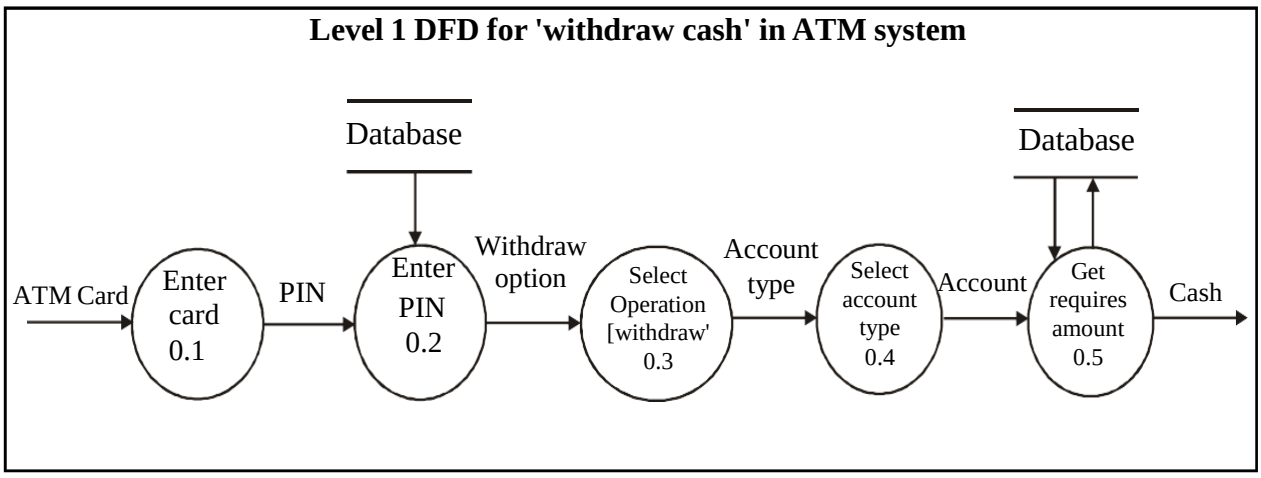
➢ **Simple DFD example of average calculator of three numbers.**

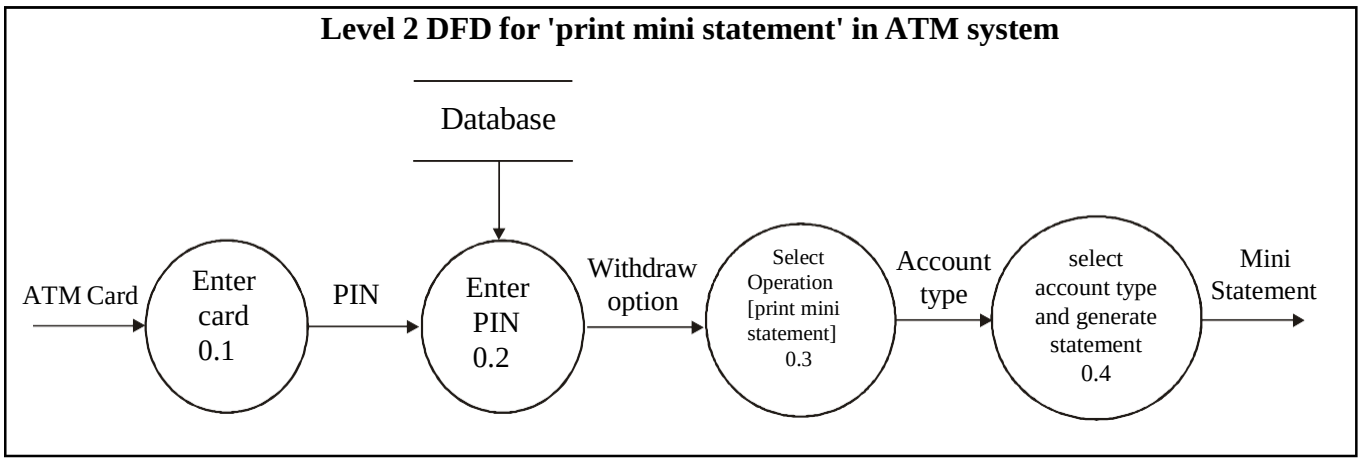


Level 0 (context diagram)



➢ **Another example of ATM system.**





Note: Level1 DFD can be drawn for each high level functional requirement shown in the SRS.

➢ Advantages of DFD model:

- DFD is a simple graphical technique which is very simple to understand and easy to use.
- It can be used as a part of system documentation.
- DFD can provide detailed description of the system components.
- It provides clear understanding to the developers about the system boundaries and analysis of the system.
- It explains the logic behind the data flow within system.
- It provides structure analysis of the system.
- Symbols used in DFD model are very less.
- It does not provide any time dependent behavior like we cannot consider at which time we have to do particular process.

➢ Disadvantages of DFD model:

- Control information is not defined by a DFD.
- DFD does not provide any specific guidance as how exactly decompose a given function into its sub functions; we have to use subjective judgment to carry out decomposition.
- Sometimes it puts programmers in little confusing state.
- Different DFD models have different symbols, e.g. in Gane and Sarson notations → process is represented as rectangle while in Demarco and Yordan notations → it is ellipse, so making confusion at the time of referring. (In some of the notations you can see the process symbol is rectangle while in some other notations, process symbol is ellipse and that is confusing).
- Physical considerations are left out in DFD.

❖ SCENARIO BASED MODELING

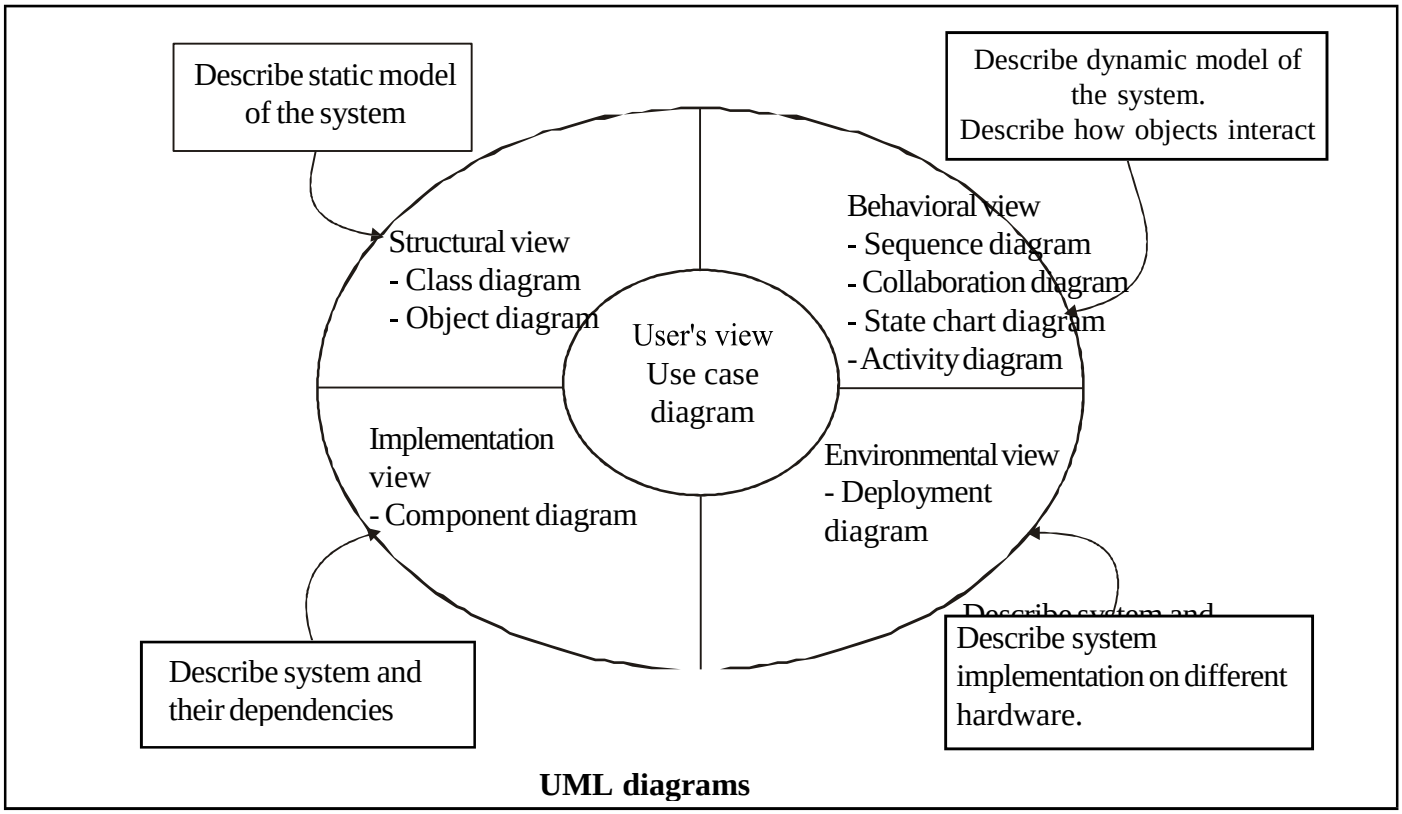
- A scenario describes a set of actions that are performed to achieve some specific conditions. And this set is specified as a sequence.
- Each step in scenario is a logically complete action performed either by the actor or by the system.

➢ UML (Unified Modeling Language):

- As the name suggests, UML is a modeling language.
- UML is a general purpose modeling language in the field of software engineering, which is designed a standard way to visualize the design of the system.
- The goal is for UML to become a common language for creating models of object oriented computer software.
- UML is very useful in documenting the design and analysis results.
- It may be used to visualize, specify, construct, and document the software system.
- UML is not a design methodology.
- UML making system easy to understand using less number of primitive symbols.

➢ UML diagrams

- UML can be used to construct nine different types of diagrams to capture five different views of a system.
- UML diagrams provide different perspectives of the software system.



UML diagrams

➢ Writing Use-Cases (Use-Case diagram):

- Use case model in UML provides system behavior.
- The use case model for any system consists of a set of "use cases".
- Use cases represent the different ways in which a system can be used by the users.

- **Use case diagram is a representation of a user's interaction with the system that shows the relationship between the users and different use cases in which user is involved.**

- The purpose of a use case is to define the logical behavior of the system without knowing the internal structure of it.
- Use case identifies the functional requirements of the system.
- Use case diagram describes "who can do what in a system".
- A use case typically represents a sequence of interactions between the user and the system.
- Use cases corresponding to the high level functional requirements.
- For example → in ATM system, the system should have following use cases.
  - Check balance
  - Withdraw money
  - Deposit money
  - Transfer money
  - Print mini statement
  - Link aadhar card
  - Pin change etc.

➢ Components of use case diagram (Representation of use case diagram)

- Three main components along with relationships are used in use case diagram.

I. Use case

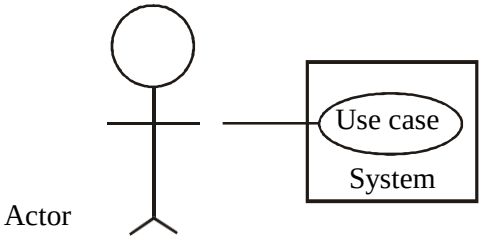
- Each use case is represented by an ellipse with the name of the use case written inside the ellipse, named by verb.
- All the use cases are enclosed with a rectangle representing system boundary. Rectangle contains the name of the system.
- It identifies, clarifies and analyzes the functional requirements of the system.

II. Actor

- An actor is anything outside the system that interacts with it, named by noun.
- Actors in the use case diagram are represented by using the stick person icon.
- An actor may be a person, machine or any external system.
- In use case diagram, actors are connected to use cases by drawing a simple line connected to it. Actor triggers use cases.

- **Each actor must be linked to at least one use case, while some use cases may not be linked to actors.**

- A simple figure showing the use cases and actor in the system.



III. Relationship

- Relationships are represented using 'a line' between an actor and a use case. It is also called communication relationship.
- An actor may have relationship with more than one use case and one use case may relate to more than one actor.

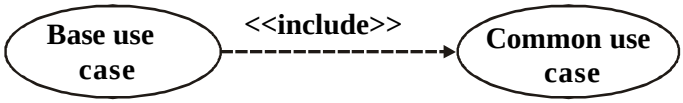
➢ **Different relationships in use case diagram are explained below:**

1. Association

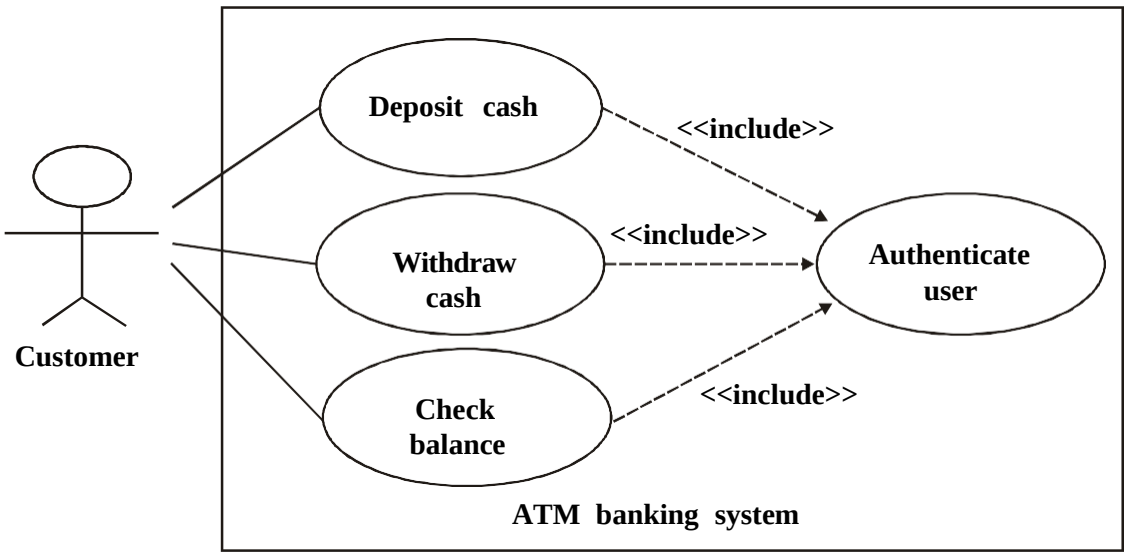
- This relationship is the interface between an actor and a use case.
- It is represented by joining a line from actor to use case.

2. Include relationship

- It involves one use case including the behaviour of another use case.
- The "include" relationship occurs when a chunk of behaviour that is similar across a number of use cases.
- It is represented using predefined stereotype <<include>>.
- An include relationship is depicted with a directed arrow having a dotted line. The tip of arrowhead points to the child use case and the parent use case connected at the base of the arrow.



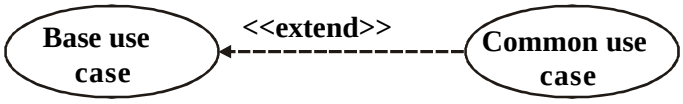
- Example of include relationship is showing in below figure.



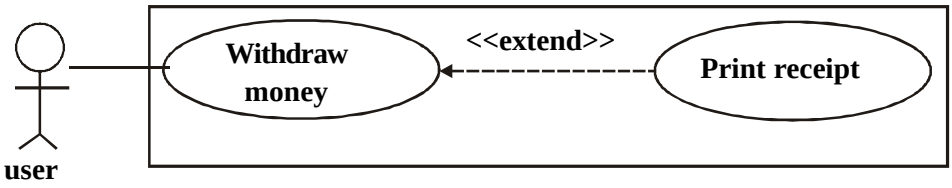
3. Extend relationship

- It allows you to show optional behavior of the system. Extend is optional and supplementary.
- This relationship among use cases is represented as a stereotype <<extend>>.
  - It is depicted with a directed arrow having a dotted line. The tip of arrowhead points to the base use case andthe child use case is connected at the base of the arrow.
  - Extend relationship exists when one use case calls another use case under certain condition (like: If - thencondition).

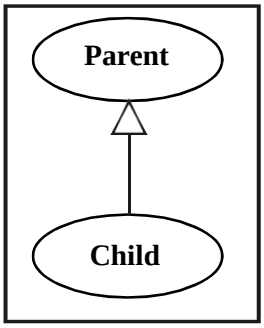
- Extend relationship is optional and supplementary.



- Example of include relationship is showing in below figure.

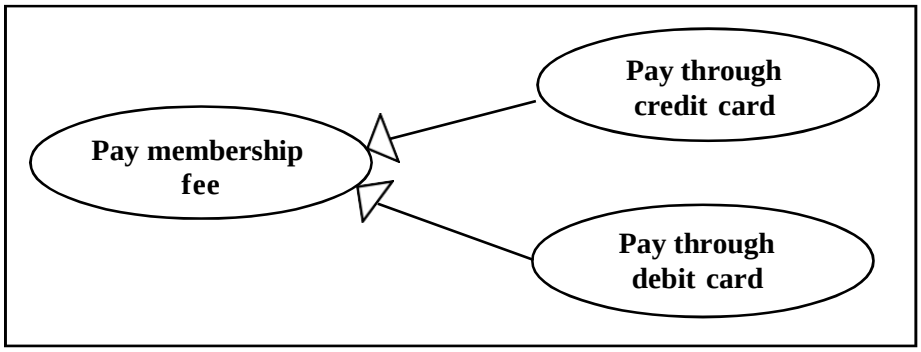


4. Generalization



- A generalization relationship is a parent-child relationship between use cases.
- Use case generalization can be used when you have one use case that is similar to another, but does slightly different.
- The child use case is an enhancement of the parent use case.
- Generalization is shown as a directed arrow with a triangle arrowhead. The child use case is connected at the base of the arrow. The tip of the arrow is connected to the parent use case.
- The child may override the behavior of its parent.

Example of generalization is showing in below figure.

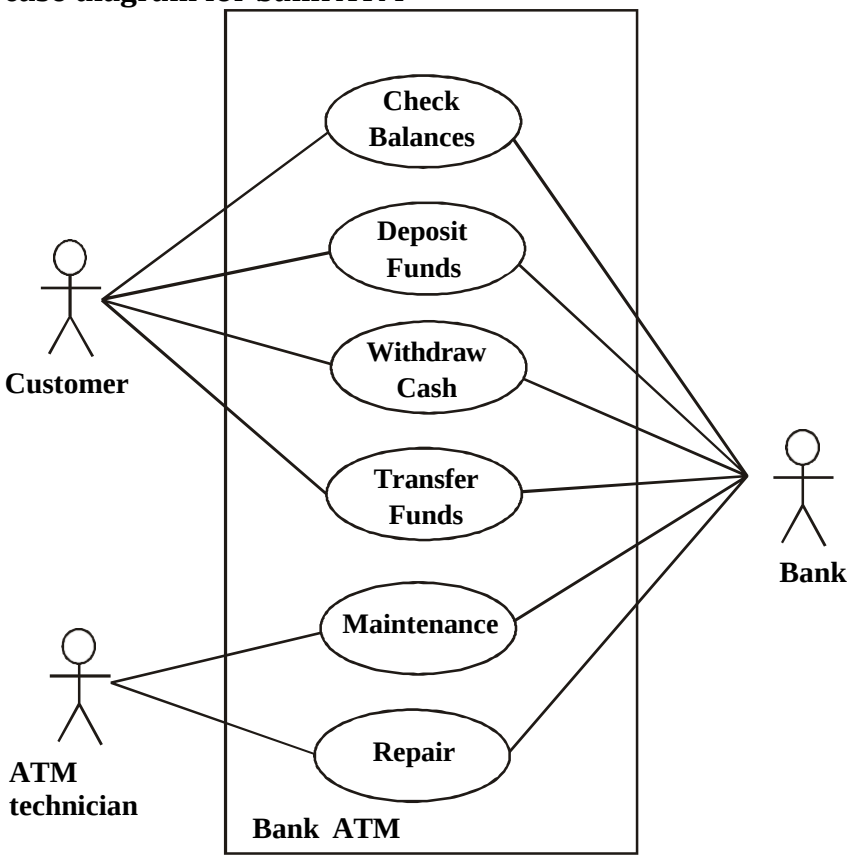


➢ Use case guidelines

- Identify all different users. Give suitable names.
- For each user, identify tasks. These tasks will be the use cases.

- Use case name should be user perspective.
- Show relationships and dependencies.

➢ **Example: Use case diagram for bank ATM**



➢ **Advantages of use case diagram**

- It is easy to understand and draw.
- It is used to capture the functional requirements of the system.
- It is used as basis for scheduling effort.
- It is used to verify whether all the requirements are captured.

➢ **Disadvantages of use case diagram**

- Use case diagrams are not fully object oriented.
- It does not provide any guideline when to stop.

➢ **Applications of use case diagram**

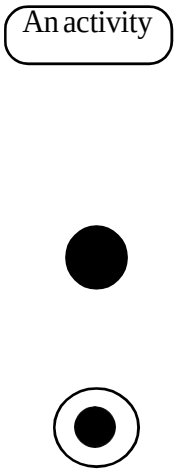
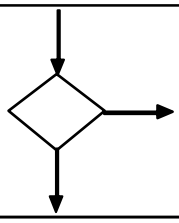
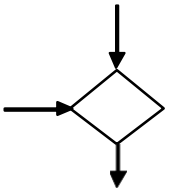
- Requirement analysis
- High level design
- Reverse engineering
- Forward engineering

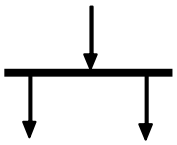
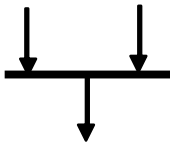
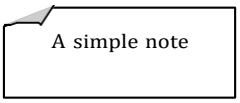
➢ **Developing an Activity diagram:**

- An activity diagram falls under the category of behavioral diagram in UML.
- An activity diagram is a UML diagram that is used to model a process. It models the actions (behavior) performed by the system components, the order in which the actions take place and conditions related to actions.

- Activity Diagrams consist of activities, states and transitions between activities and states.
- It describes how the events in a single use case relate to one another.
- The aim of activity diagram is to record the flow of control from one activity to another of each actor and to show interaction between them.
- It mainly represents series of actions and flow of control of a system.
- It focuses on the how of activities involved in a single process. Also shows how activities depend on one another.
- Activity diagrams represent workflows in a graphical way.
- Activity diagrams are similar to procedural flow charts. The difference is that activity diagram support parallel activities.
- An activity is a state with an internal action and one or more outgoing transitions.
- An interesting feature of the activity diagrams is the swimlanes. It enables you to group activities based on who is performing them. So, swimlanes make group of activities based on actors.

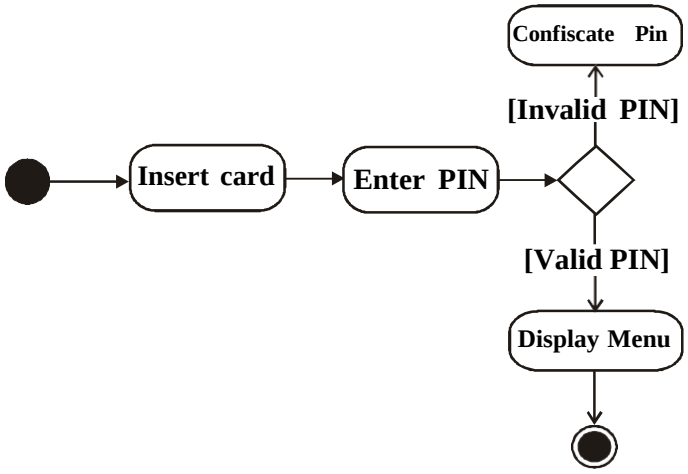
➢ Elements (components) of an activity diagram

Elements or Components and its description	Symbol
Activity – It represents a particular action taken in the flow of control. – It is denoted by a rectangle with rounded edges. And labelled inside it describing corresponding activity. – There are two special type of activity nodes : **1. Initial activity (OR Start activity)** – This shows the starting point or first activity of the flow. – It is denoted by a solid circle. **2. Final activity (OR End activity)** – The end of the activity diagram shown by a bull's eye symbol. It represents the end point of all activities.	
Flow or Transition – A flow (also termed as edge or transition) is represented with a directed arrow. – This is used to show transfer of control from one activity to another.	
Decision (Branch) – A decision node represented with a diamond. – It is a branch where single transition (flow) enters and several outgoing transitions.	
Merge – This is represented with a diamond shape with two or more input transitions and a single output transition.	

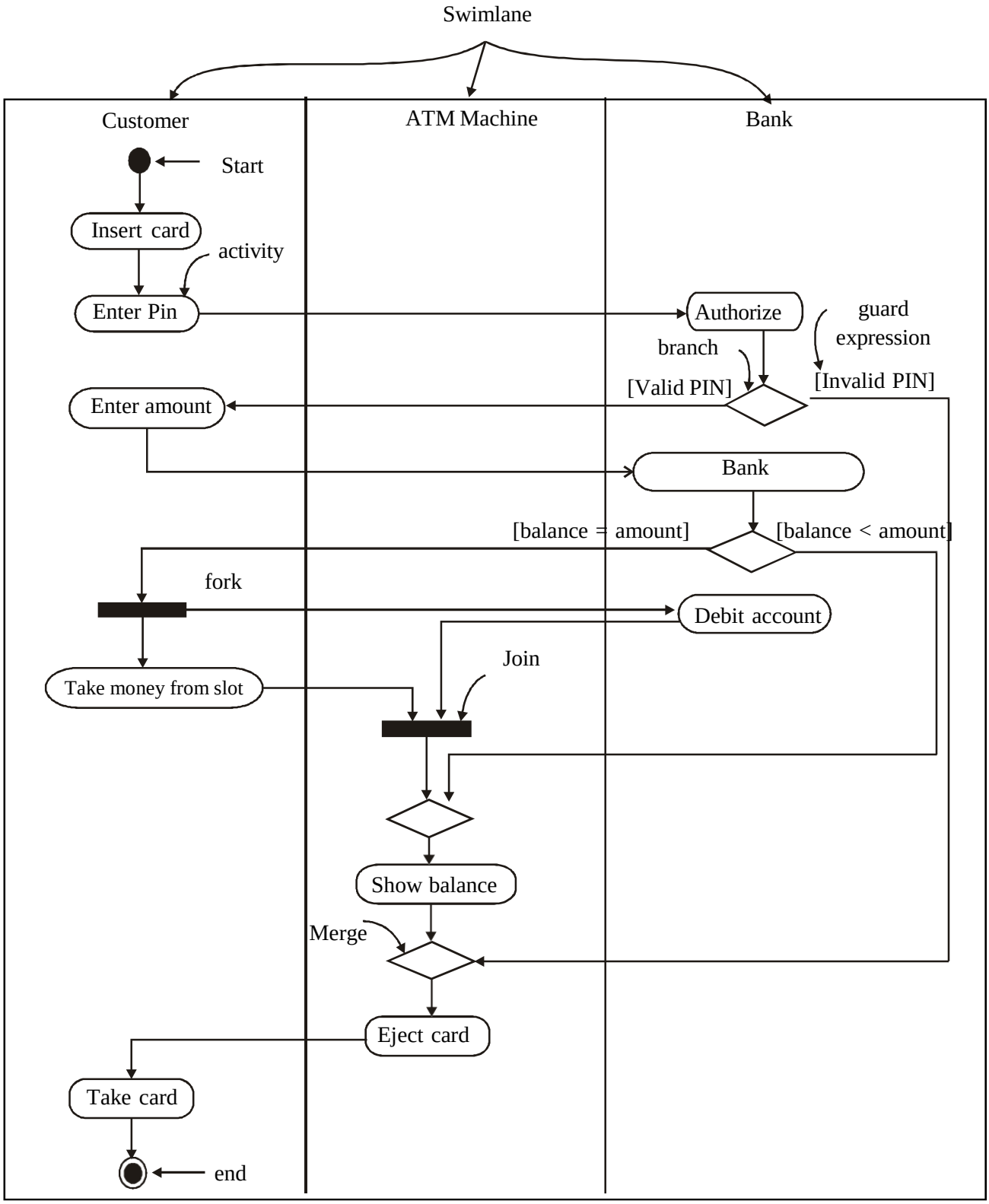
Fork – Fork is a point where parallel activities begin. – Fork is denoted by black bar with one incoming transition and several outgoing transitions. – When the incoming transition is triggered, all the outgoing transitions are taken Into parallel.	
Join – Join is denoted by a black bar with multiple incoming transitions and single outgoing transition. – It represents the synchronization of all concurrent activities.	
Note – UML allows attaching a note to different components of diagram to present some textual information. – It could be some comments or may be some constraints. – A note generally attached to a decision point to indicate the branching criteria. – It is denoted by a rectangle with cut a side.	
Partition or Swimlanes – Different components of an activity diagram can be logically grouped into different areas, called partition or swimlanes. – They often correspond to different users or different units of organization. – It is denoted by drawing vertical parallel lines. – Partitions in an activity diagram are not mandatory.	
Guard conditions – Guard conditions control transition from alternative transitions based on condition. – These are represented by square brackets.	**[ ]**

Typical symbols used in activity diagram

- **A simple example activity diagram (ATM system)**



- **Another example showing swimlanes**



- **Advantages of activity diagram**
  - Activity diagrams can be very useful to understand complex processing activities.
  - Different activities are grouped together based on actor. That is represented by swimlanes.
  - It can be useful for analyzing a use case and understating workflow of system.

- Activity Diagrams are good for describing synchronization and concurrency between activities.
- Partitioning can be helpful in investigating responsibilities for interactions and associations between objects andactors.

➢ **Disadvantages of activity diagram**

- The activity diagram does not provide message part. Means do not show any message flow from one activityto another.
- It can't describe how objects collaborate.
- It takes time to implement.
- Complex conditional logics (like Truth table) can't be represented by activity diagram.
- There are lot many symbols compare to other UML diagrams, so sometimes make it confusing to the developer.

➢ **When to use Activity diagram (Applications of activity diagram)**

- Mainly activity diagram used to describe the parallel behavior of the system. It makes a great tool for workflowmodeling.
- It is also used in multithreaded programming application.

➢ **Difference between flowchart and activity diagram**

Flow chart	Activity diagram
It is limited for sequential access.	It is used for parallel and concurrent processing.
It is used for flow of control through an algorithm, not used for object oriented procedure.	It is usually used for object oriented systems.
Concept of swimlanes is not there in it.	It has the functionality of swimlanes.
It has limited functionalities compare to activity diagram.	It has more functionality.

Unit-4

4.1 RESPONSIBILITIES OF SOFTWARE PROJECT MANAGER

- Software project managers take the overall responsibility of project success.
- A project manager is the person who is responsible for accomplishing (પૂર્ણ કરવું) the stated project objectives.
- The job responsibility of a project manager ranges from invisible activities like building up team spirit to highly visible customer presentations. (Planning to Deployment).
- A project manager bridging the gap between the production team and client.
- He managing the constraints of the project management triangle, which are cost, time, scope, and quality.
- General activities of manager like → project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, interfacing with clients, managerial report writing and presentations, etc.
- All the above activities are mainly classified into: *project planning, project monitoring and control activities.* Project planning activity starts before development, while monitoring and control starts after development.
- Key among his or her duties is the recognition of risks that affect the success of the running project (risk management). It follows that a project manager is one who is responsible for making decisions both large and small, in such a way that risk is controlled and minimized.
- Time and cost estimation are also important factors of project manager responsibilities.
- Follow project status and modify it to ensure success.
- Every decision taken by the project manager should be taken in such a way that it directly benefits the project.
- He sets up development milestones and entry or exit criteria.

✢ **The skills required in project manager to manage the project, are :**

- He must have theoretical knowledge of different project management techniques.
- A good decision making capabilities also required in project manager.
- He should be client representative and has to determine and implement the exact needs of the client, and capable of understanding and discussing the problems with customers.
- He should have the management skills like ask the questions and resolve conflicts regarding project.
- Successful project manager one who focuses on risk management.
- He should have team leadership skill so the project which is divided into different persons are managed well and completed as per schedule.
- He should have the experience in the related area of the developing project.
- Monitoring and scheduling the progress of the project.
- Evaluating performance of each milestone of the project.
- Some skills like tracking and controlling the progress of the project, customer interaction, managerial presentations, and team building are acquired through experience.

4.2 METRICS FOR PROJECT SIZE ESTIMATION

- Metrics are the tools that help in better monitoring and control.
- Size of the program (or project) is neither the number of bytes that the source code occupies nor the byte size of the executable code.
- But it is an indicator of the effort and time required to develop the project. So, it indicates project development complexity.
- The project size is a measure of the problem complexity in terms of the effort and time required to develop the entire product.
- There are several metrics to measure problem size. Each of them has its own advantages and disadvantages.
- We will consider two important metrics to estimate size : *Lines of code (LOC) and Function point (FP).*

4.2.1 Lines of code (LOC) :

- The simplest measure of problem size is Lines Of Code (LOC) or Source Lines Of Code (SLOC).
- It is very popular because of its simplicity.
- LOC is a software metric used to measure the size of a software program by counting the number of lines in the text of the program's source code. Comments line and headers lines are ignored at counting the source code.
- In other term, it is software metric that measures the size of software in terms of lines in the program.
- In order to estimate the LOC count at the beginning of a project, project managers usually divide the problem into modules and each module into sub modules and so on, until the sizes of the different leaf-level modules can be approximately predicted.
- According to Boehm, every line of source text should be calculated as one LOC.
- By using the estimation of the lowest level modules, project managers arrive at the total size estimation.
- One main advantage of LOC is that it is very simple to understand and directly relates to end product.

✢ **However, LOC has several disadvantages.**

- LOC is language dependent. LOC focuses on coding activity only, while a good problem size measure should consider the overall complexity of the problem.
- LOC gives only numerical values of problem size, that is depend on coding style. It can't measure the size of specification.
- LOC does not measure with the quality and efficiency of the code.
- LOC metric suffer from accuracy when use of high level languages, code reuse, library subroutine etc.
- LOC doesn't issue logical and structural complexities.
- It is very difficult to accurately estimate the LOC in the final product from the problem specification. Accurate LOC can be computed only after code has been fully developed.
- LOC doesn't work well with non-procedural languages.

4.2.2 Function Point metric (FP) :

- Function point metric was first proposed by Albrecht in 1979 and internationally approve modified model in 1983. This metric overcomes many of the shortcomings of the LOC metric.
- Function point metrics, measure functionality from the users' point of view, that is, on the basis of what the user requests and receives in return.

- One of the important advantages of using the function point metric is that it can be used to easily estimate the size of a software product directly from the problem specification.
- In LOC metric, the project size can be accurately determined only after the product has fully been developed. While in case of FP, the size of a software product is directly dependent on the number of different functions or features it supports.
- Software supporting many features or functions should have larger FP in size.
- Each function when invoked reads some input data and transforms it to the corresponding output data.
- By using the number of input and output data values, function point metric computes the size of a software product.
- FP considers five different characteristics of the product to calculate the size. The function point (FP) of given software is the weighted sum of these five items, and it will give unadjusted function point (UFP).

  UFP = (Number of inputs)*4 +
  (Number of outputs)*5 +
  (Number of inquiries)*4 +
  (Number of files)*10 +
  (Number of interfaces)*10
- The meaning of each parameter is as follow :

(1) Number of inputs

- Each data item input by the user is counted.
- Group of related inputs are considered as a single input.
- For example, while entering the data concerning student to student information system software; the data items name, age, gender, address, phone number, etc. are together considered as a single input.

(2) Number of outputs

- It refer to reports printed, screen outputs, error messages produced, etc.
- The set of related data items is counted as one input.

(3) Number of inquiries

- It is the number of distinct interactive queries which can be made by the users.
- These should be user commands for specific actions.

(4) Number of files

- Each logical file is counted. A logical file means groups of logically related data is counted as a file.
- The files can be data structures or physical files.

(5) Number of interfaces

- The interfaces used to exchange information with other systems.
- Examples of such interfaces are data files on tapes, disks, communication links with other systems etc.
- Once the unadjusted function point (UFP) is computed, the technical complexity factor (TCF) is computed next.
- TCF refines the UFP by considering 14 factors which assigns 0 (no influence) to 6 (strong influence). These numbers are summed and yielding DI (Degree of Influence).
- Now TCF is computed as

$$\mathbf{TCF = (0.65 + 0.01 * DI)}$$

- As DI can vary from 0 to 70, TCF can vary from 0.65 to 1.35.

Finally: FP = UFP * TCF

✢ **Advantages of function point (FP)**

- It is not restricted to code.
- It is language independent.
- It is more accurate than estimate LOC.
- We can have the measure from the specification that can't be done with LOC.

✢ **Shortcomings of function point (FP) metric**

- It ignores quality of output.
- It is fully oriented to traditional data processing system.
- Major shortcoming of FP is it does not calculate the algorithmic complexity of software.
- Function point has been criticized as not being universally applicable to all types of software. Because FP doesn't capture all functional characteristics of real-time software.
- The FP is originally designed for business information system, and it is not suited for many engineering and embedded systems.
- To overcome the problem of FP, an extension of the function point metric called feature point metric is proposed.
- Feature point metric incorporates an extra parameter algorithm complexity.
- This parameter ensures that the computed size using the feature point metric shows the fact that the more is the complexity of a function, the greater is the effort required to develop it and therefore its size should be larger.

4.3 PROJECT ESTIMATION TECHNIQUES

☛ **The Estimation is prediction or a rough idea to determine how much effort would take to complete a defined task.**

- Project estimation is a very important activity.
- The important project parameters that are estimated include: project size, effort required to develop the software, project duration, and cost.
- Project estimation determines how much money, effort. Resources and time it will take to build a specific system or product.
- These estimates help in project resource planning and scheduling.
- There are three main categories of project estimation techniques:
  1. Empirical estimation techniques
  2. Heuristic techniques
  3. Analytical estimation techniques

4.3.1 Empirical (અનુભવો પર આધારિત) estimation techniques :

- These techniques do not use mathematical equations to estimate the cost of the project. As it is done using prior experience and common sense over the years.
- Empirical estimation techniques are based on making an educated guess of the project parameters.

- The project managers use their past experiences to make guess.
- There are two popular empirical estimation techniques :
  1. Expert judgment
  2. Delphi cost estimation

1. Expert judgment

- This is most widely used empirical estimation technique.
- In which, an expert makes an educated guess of the problem size after analysing the problem in detail.
- Expert estimates the costs of different modules (or units) then combines them for overall estimation.
- But in some situation, an expert may not have knowledge of all aspects of project or he may overlook some factors. For simple example, the expert may have very good knowledge of database and analysis part but may not be aware with testing or presentation part.
- So, the chance of human error or individual bias should be there in this technique.
- To solve the above problem $\rightarrow$ a more refined form of expert judgment is the estimation made by group of experts. It should minimize the factors of above problem.
- Even after this, chance of biasing is also there in expert group judgment also. Because sometimes unusual decision taken by one member can affect the whole group.

2. Delphi cost estimation

- It provides solution to the shortcomings of the expert judgment approach.
- Delphi estimation is carried out by a team having a group of experts (estimators) and a coordinator.
- In it, the coordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate.
- Estimators complete their individual estimates anonymously (unidentified) and submit to the coordinator.
- The estimators mention any unusual characteristics of the product/project in their estimation report.
- The coordinator prepares and distributes the summary of the responses of all the estimators and include any unusual noted by any estimator.
- Based on this summary, the estimators re-estimate.
- This process is iterated for several rounds.
- Discussion among the estimators is not allowed during the entire estimation process.
- After the completion of several iterations of estimations, the coordinator takes the responsibility of compiling the results and preparing the final estimate.

4.3.2 Heuristic estimation techniques :

- In these techniques, project parameters are modelled using mathematical expressions.
- Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the value of the basic parameters in the mathematical expression.
- Various heuristic estimation models can be divided into the following two classes: static single variable model and static multi variable model.
- In single variable estimation models, estimate the desired characteristics of a problem such as software product size.

- A single variable estimation model takes the following form :

Estimated Parameter = c1 * ed1

(where e is the basic characteristic of the software (independent variable) already estimated, Estimated Parameter is the dependent parameter to be estimated, c1 and d1 are constants.)

- The example of static single variable cost estimation variable is the basic COCOMO model.
- The multivariable cost estimation model takes the following form :

Estimated Parameter = c1*e1d1 + c2*e2d2+ ...

(where e1, e2… are the basic characteristic of the software (independent variable) already estimated and c1, c2, d1, d2, … are constants.)

- Multivariable estimation models are expected to give more accurate estimates compared to the single variable models.
- The example of a multivariable estimation model is the intermediate COCOMO model.
- ✢ After analysing various types of projects size using LOC, Boehm postulated that, any software development project can be classified into one of the following three categories based on the development complexity :
  - (i) Organic
  - (ii) Semi detached
  - (ii) Embedded
- This classification is based on characteristics of the product, development team and development environment.
- These three product classes correspond to application, utility and system programs, respectively.
- Data processing programs are considered to be application programs. Compilers, linkers, etc., are utility programs. Operating systems and real-time system programs, etc. are system programs.

4.3.3 Analytical estimation techniques :

- This Technique derives the required results starting from certain simple assumptions.
- Analytical estimating is a structured estimating technique often used in work measurement.
- In this technique, the required results are derived with basic assumptions regarding the project.
- Thus, unlike empirical and heuristic techniques, analytical techniques do have scientific basis.
- Halstead's software science is an example of an analytical technique which is especially useful for estimating software maintenance efforts.
- However, Halstead's software science is not very useful for planning development projects because this technique derives the required results after program has been developed. (It depends on completed code). But it is very useful in estimation software maintenance efforts.
- So, for predicting software estimation, it outperforms both empirical and heuristic techniques.

☛ **Boehm's definition of organic, semidetached, and embedded systems are explained below.**

(i) Organic

- This type of project has small group of team.
- Develop well understood application programs and having experienced developers with similar types of programs development.
- Little or no innovation is there in this type of projects.

- Project size should be 2-50 KLOC in organic type.
- For example data processing programs, payroll, inventory management system.

(ii) Semidetached

- This type of development consists of a mixture of experienced and inexperienced staff.
- Team members may have limited experience on related systems.
- Medium innovation is there in this type of projects.
- Project size should be 50-300 KLOC in organic type.
- For example compilers, interpreters, assemblers.

(iii) Embedded

- This type of software strongly coupled to complex hardware.
- Tight or inflexible (rigid) regulations on the operational procedures exist.
- In this projects, great deal with innovation and requirements constraints. Deadlines are strict and development consists of complex interface.
- Project size is large and development team is also large among which few are experienced with same type of development.
- Project size is approximately over 300 KLOC.
- For example real time system like air traffic control, ATMs etc.
- Brooks states that utility programs are three times as difficult to write as application programs and system programs are three times as difficult as utility programs. So the relative level of complexity ratio for these three categories of products is 1:3: 9.
- For the three product categories, Boehm provides different sets of expressions to predict the effort (in unit of person-months) and development time from the size estimation given in KLOC.

4.3.4 COCOMO (A Heuristic Estimation Technique) :

- COCOMO (Constructive COst estimation MOdel) was proposed by Boehm.
- It is regression based model provides hierarchy of software cost estimation models.

☛ (**Note:** *regression method* used to identify the relationship between one dependent variable and several independent variables.)

- Software cost estimation should be done through three stages: Basic COCOMO, Intermediate COCOMO, and complete COCOMO.
- Each level achieves successive accuracy.

✢ Basic COCOMO model

- It is static, single valued model that computes software development efforts using program size (expressed in LOC).
- It is quick and rough estimation technique.
- The basic COCOMO model gives an approximate estimate of the project parameters.
- The basic COCOMO estimation model is given by the following expressions:

$$\textbf{Effort (E) = a1 * (KLOC)a2 PM}$$

$$\textbf{Tdev = b1 * (Effort)b2 Months}$$

Where :

KOLC is Kilo Lines of Code.

a1, a2, b1, b2 are constants for each category of software products.

Tdev is the estimated time to develop the software, expressed in months.

Effort is the total effort required to develop the software product, expressed in person months (PMs).

- The effort estimation is expressed in units of person-months (PM).

➠ **Estimation of development effort in basic COCOMO model.**

- For the three classes of software products, the formulas for estimating the effort based on the code size are shown below :

Organic **:** Effort (E) = $2.4 (KLOC)^{1.05}$ PM

Semidetached : Effort (E) = $3.0 (KLOC)^{1.12}$ PM

Embedded **:** Effort (E) = $3.6 (KLOC)^{1.20}$ PM

➠ **Estimation of development time in basic COCOMO model**

- For the three classes of software products, the formulas for estimating the development time based on the effort are given below :

Organic **:** Tdev = $2.5 (effort)^{0.38}$ Months

Semidetached : Tdev = $2.5 (effort)^{0.35}$ Months

Embedded **:** Tdev = $2.5 (effort)^{0.32}$ Months

- It is important to note that the effort and the duration estimations obtained using the COCOMO model are called as nominal effort estimate and nominal duration estimate.

☛ **(Note : મિત્રો, exam માં જો Basic COCOMO મોડેલનું example પુછાય તો તમારે ઉપર જણાવેલ constants જેવા કે a1, a2, b1, b2 ની values યાદ રાખવી પડે. જેને સરળતાથી યાદ રાખવા માટે નીચે આપણે એક ટેબલ બનાવેલું છે.)**

Basic COCOMO Model				
Type of System	a1	a2	b1	b2
Organic (2-50 KLOC)	2.4	1.05	2.5	0.38
Semi-detached (50-300 KLOC)	3.0	1.12	2.5	0.35
Embedded (Approx over 300 KLOC)	3.6	1.20	2.5	0.32

Example : Assume that the size of an organic type software product has been estimated to be 2,000 lines of source code. Assume that the average salary of software engineers be Rs. 20,000/- per month. Determine the effort required to develop the software product and the nominal development time.

Solution : Given problem is in basic COCOMO model and source code lines are 2000 (means 2 KLOC), so it is coming in organic type of development.

For the basic COCOMO model :

Effort(E) = $2.4 * (KLOC)^{1.05}$ PM

= $2.4 * (2)^{1.05}$

= 5 PM

Nominal development time = $2.5 * (effort)^{0.38}$ Months

= $2.5 * (5)^{0.38}$

= 4.6 Months

Cost required for developing the product : = $4.6 * 20000$

= 92000 /-

✢ **Intermediate COCOMO model**

- The basic COCOMO model assumes that effort and time development are functions of the product size alone. Its result in lack of accuracy.
- But some factors from other projects may also affect the efforts and time.
- Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account.
- The intermediate COCOMO model doing this by using a set of 15 cost drivers (multipliers) based on various attributes of software development.
- So, Intermediate COCOMO model - computes software development effort as function of program size and a set of "cost drivers" that include subjective assessment of product, hardware, personnel and project attributes.
- Project manager doing the task of rating these 15 parameters in scale 1 to 3 and then cost driver values are estimated.
- The cost drivers can be grouped into four categories.

1. Product attributes :

- Required software reliability (RELY)
- Database size (DATA)
- Product complexity (CPLX)

2. Computer attributes :

- Execution time constraint (TIME)
- Main storage constraint (STOR)
- Virtual machine volatility (VIRT)
- Computer turnaround time (TURN)

3. Personal attributes :

- Analytical capability (ACAP)
- Application experience (AEXP)
- Programmer capability (PCAP)
- Virtual machine experience (VEXP)
- Programming language experience (LEXP)

4. Project attributes :

- Modern programming practice (MODP)
- Use of software tools (TOOL)
- Requirement development schedule (SCED)

✢ **Complete COCOMO model**

- It is also known as detailed COCOMO model or advanced COCOMO model.
- A major shortcoming of both the basic and intermediate COCOMO models is that they consider a software product as a single homogeneous entity.
- But most large systems are made up several smaller sub-systems.
- For example, some subsystems may be considered as organic type, some semidetached, and some embedded.
- And these subsystems may be different requirements, development complexities and may not have previous experience of similar developing.
- The complete COCOMO model considers these differences in characteristics of the subsystems.
- Detailed COCOMO model - incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process.
- It estimates the effort and development time as the sum of the estimates for the individual subsystems.
- The cost of each subsystem is estimated separately.
- This approach reduces the margin of error in the final estimate.
- For example a distributed Management Information System (MIS). Because it can have following sub components :
  - ➠ Graphical User Interface (GUI) part (organic)
  - ➠ Database part (semi detached)
  - ➠ Communication part (embedded)
- The costs for these three components can be estimated separately, and summed up to give the overall cost of the system.

✢ **Advantages of COCOMO model**

- It is easy to use.
- It is repeatable process.
- It is versatile enough to support different modes and levels.
- It works well on projects that are not so different in size, process and complexity.
- It is highly standardized, based on previous experience.
- It can be detailed documented.

✢ **Disadvantages of COCOMO model**

- COCOMO models are not accurate and not good for scientific justification.
- It ignores software safety issues.
- It ignores personal turnover levels.
- All the levels of COCOMO are dependent on size estimates.
- It also ignores many hardware issues.

4.4 SCHEDULING

- Project-task scheduling is an important project planning activity. It involves deciding which tasks would be taken up when.
- Project scheduling is a tool that distributes estimated efforts across the project duration and allocates these efforts to specific tasks.
- In order to schedule the project activities, a software project manager needs to do the following :
  - ➠ Identify all the tasks needed to complete the project.
  - ➠ Break down large tasks into small activities.
  - ➠ Determine the dependency among different activities.
  - ➠ Establish the most likely estimates for the time durations necessary to complete the activities.
  - ➠ Allocate resources to activities.
  - ➠ Plan the starting and ending dates for various activities.
  - ➠ Determine the critical path.
- Among all of the above activities: identifying tasks and breaking down them into small activities done through work breakdown structure (WBS). PERT and CPM used to sequence the activity and estimating time and project schedule is developed through Microsoft project tool.

4.4.1 Work breakdown structure (WBS) :

- Work Breakdown Structure (WBS) is used to decompose a given task set recursively into small activities.
- The WBS is a uniform, consistent, and logical method for dividing the project into small, manageable components for purposes of planning, estimating, and monitoring.
- An effective WBS encourages a systematic planning process, reduces the omission of key project elements, and simplifies the project by dividing it into manageable units.
- A WBS will provide a roadmap for planning, monitoring, and managing all factors of the project like ' resource allocation, scheduling, budgeting, productivity, performance etc.
- WBS can be shown graphically in a hierarchical tree structure and developed top to bottom manner.
- The root of the tree is labelled by the problem name.
- Each node of the tree is broken down into smaller activities that are made the children of the node.
- Each activity is recursively decomposed into smaller sub-activities until at the leaf level.
- WBS can be done by the decision of the project manager.

✢ **Types of WBS :**

- There are three types of WBS as follows :

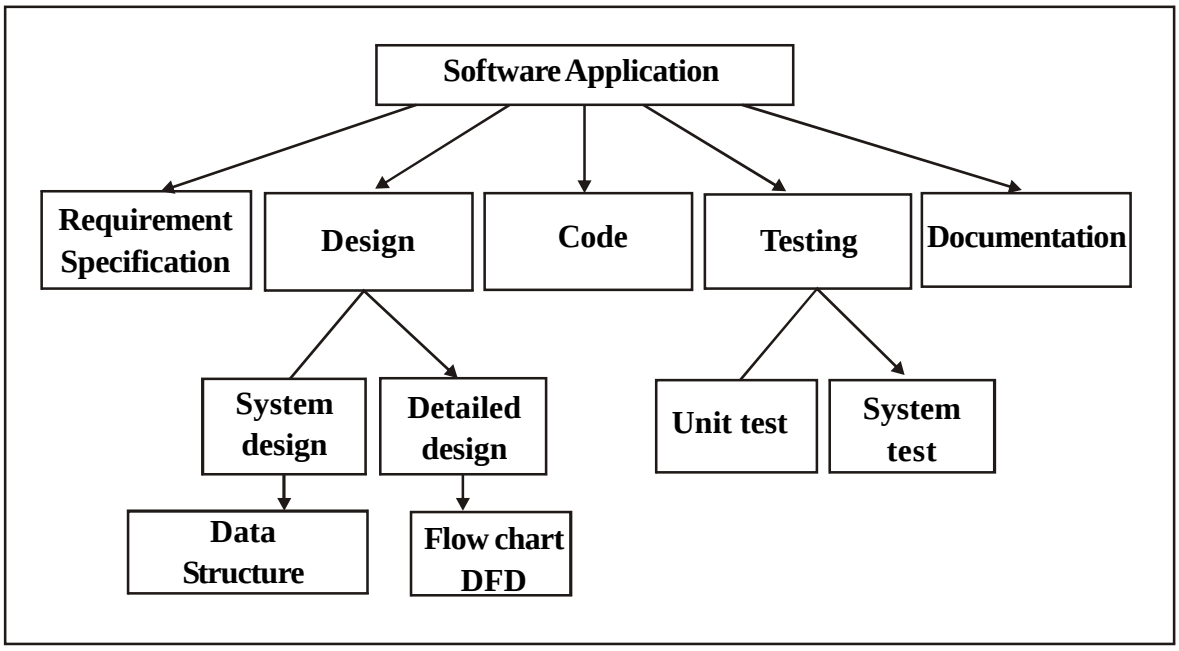
(i) Process WBS : it decomposes large processes into smaller ones. Each process finally decomposed in the task.

(ii) Product WBS : it decomposes large entity into smaller components. It is used by system engineers.

(iii) Hybrid WBS : in includes both process and product elements into single WBS.

- There are two methods of WBS presentation :

1. Tree structure :



2. Indented list form :

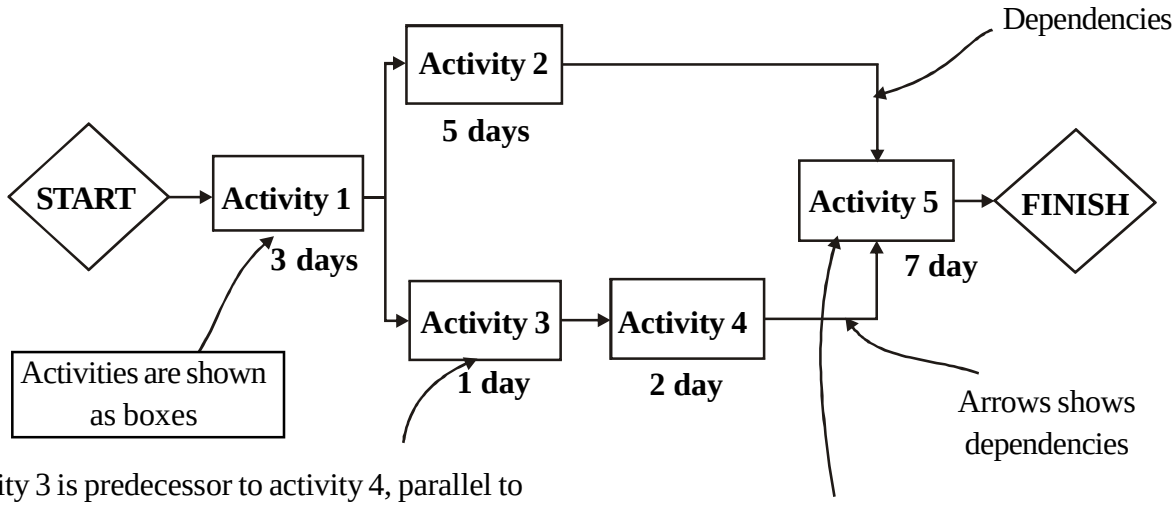
Level	Tasks
0	Top level Process (Product) (Software application)
1	Requirement Specification
2	Design 2.1 System Design 2.1.1 Data structure 2.2 Detailed design 2.2.1 Flow chart, DFD
3	Code
4	Testing 4.1 Unit test 4.2 System test
5	Documentation

✢ **Disadvantages of WBS**

- It specifies tasks only, not the process by which task is carried out.
- It doesn't specify the person who is doing the task.
- WBS is project specific, so inter project comparison is difficult.
- In WBS, only the list of tasks and deliverables are prescribed, not how the particular work will be completed.
- WBS doesn't provide any sequence or plan for the tasks.

4.4.2 Activity network :

- In project management, an activity is a task that needs to be accomplished within defined period of time or by deadline to achieve goal.
- Activities in a project are graphically represented using activity network diagram.
- Activity network is a network graph using nodes with interconnecting edges to represent tasks and their planned sequence of completion, interdependence and interrelationship that must be accomplished to reach the project goals.
- An Activity Network Diagram helps to find out the most efficient sequence of events needed to complete any project. It enables you to create a realistic project schedule by graphically showing :
  - The total amount of time needed to complete the project
  - The sequence in which tasks must be carried out
  - Which tasks can be carried out at the same time
  - Which are the critical tasks that you need to keep an eye on.
- At the time of drawing activity network diagram, each activity is represented by a rectangular node and the duration of the activity is shown alongside each task.



Activity 3 is predecessor to activity 4, parallel to activity 2 and successor to activity 1.

Activity 5 cannot begin until activity 2 and 4 have been completed.

- The Activity Network diagram displays interdependencies between tasks through the use of boxes and arrows. Arrows pointing into a task box come from its predecessor tasks, which must be completed before the task can start. Arrows pointing out of a task box go to its successor tasks, which cannot start until at least this task is complete.
- **Activity Network Diagram Drawing Rules :**
  - All the preceding activities must be completed before the project completed.
  - The arrows represent the logical precedence of the project.
  - The task which is dependent on other tasks cannot start until the tasks on which it depended are completed.

✢ **Applications**

- Activity Networks and PERT (Programming Evaluation and Review Technique) Charts are typically used to document complex projects in a visual manner.
- They are also used to establish the critical path of a project.

4.4.3 Critical Path Method (CPM) :

- The CPM method was discovered by M.R.Walker in 1957. Critical path method is a network analysis technique.
- Critical path is the sequence of activities with the longest duration. A delay in any activity on this path will result in a delay for the whole project. The activities on the critical path are critical activities.
- CPM used to calculate project completion time.
- CPM used to predict the project duration by finding out sequence of activities has the least amount of scheduling flexibilities.
- The project manager identifies the critical activities of the project.
- CPM deals with both cost and time.
- CPM is used to calculate expected completion time of the project.
- It is based on single time estimation.

✢ **Need of CPM**

- Planning resource requirements.
- Control resource allocation.
- Prediction of deliverables.
- Internal and external program review.
- Performance evaluation.

✢ **Advantages**

- It provides clear, concise and unambiguous way of documenting project plans, schedules, time and cost.
- It is mathematically easy and simple.
- It is useful to new project managers.
- It displays dependencies which help in scheduling.
- It determines slack time. Which is the total time for that task may be delayed to complete.
- It can display parallel running activity.
- It is widely used in industry purpose.

✢ **Disadvantages**

- It is too complex for large projects.
- It doesn't handle the scheduling of people and resource allocation.
- Critical path should be calculated carefully.
- Calculation of estimating the completion time is difficult.
- Activity time estimates are subjective and depend on judgment.

✢ **Applications**

- Used in construction activities.
- Used in medical and surgical sector.
- Used in transportation activities and oil refineries.

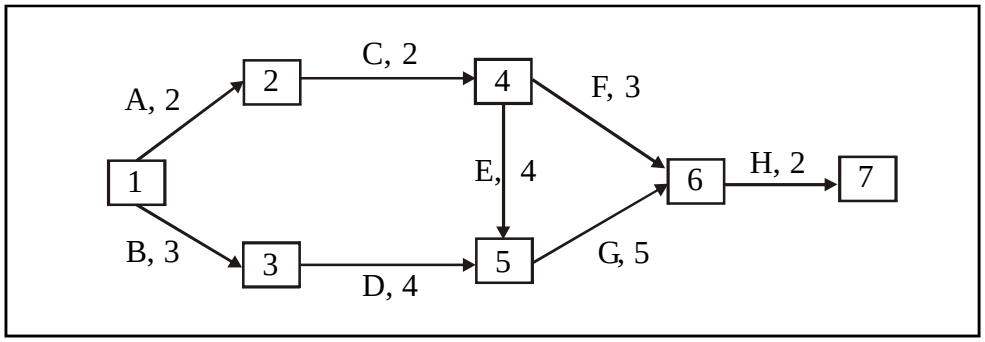
✢ **Example**

- Find critical path (project completion time) for the air pollution control system having different activities listed below :

Activity	Description	Time	Immediate Predecessor
A	Build internal components	2	None
B	Modify roof and floor	3	None
C	Construct collection stack	2	A
D	Pour concrete and install frame	4	B
E	Build high-temperature burner	4	C
F	Install control system	3	C
G	Install air pollution device	5	D,E
H	Inspection and testing	2	F,G

- For above problem, first we have to draw the activity network diagram.
- In that diagram each activity is listed on arc. Activities are arranged according to precedence given above.

☛ (This type of network diagram is called **Activity on Arc (AoA),** Activity on Nodes (AoN) is also another form of network diagram which is drawn at the end of this chapter.)

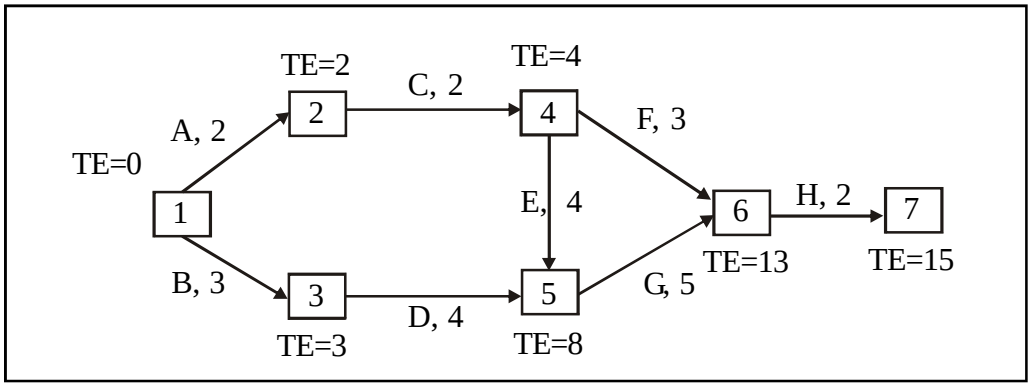


- To process for CPM, we have to compute earliest completion time (TE) and Latest completion time (TL) for each node. Earliest completion time (TE) is calculated in forward pass and Latest completion time (TL) is calculated in backward pass.

TE(Node 1) = 0

TE(Node i) = max{t(p)}, $i \neq 1$

Where t(p) denotes the sum of time durations for a path p and where the maximum is taken overall path from node 1 to node i. when we assign a value to the sink node (last node), this value is the earliest completion time for the entire project. Now calculate TE for the above network diagram.

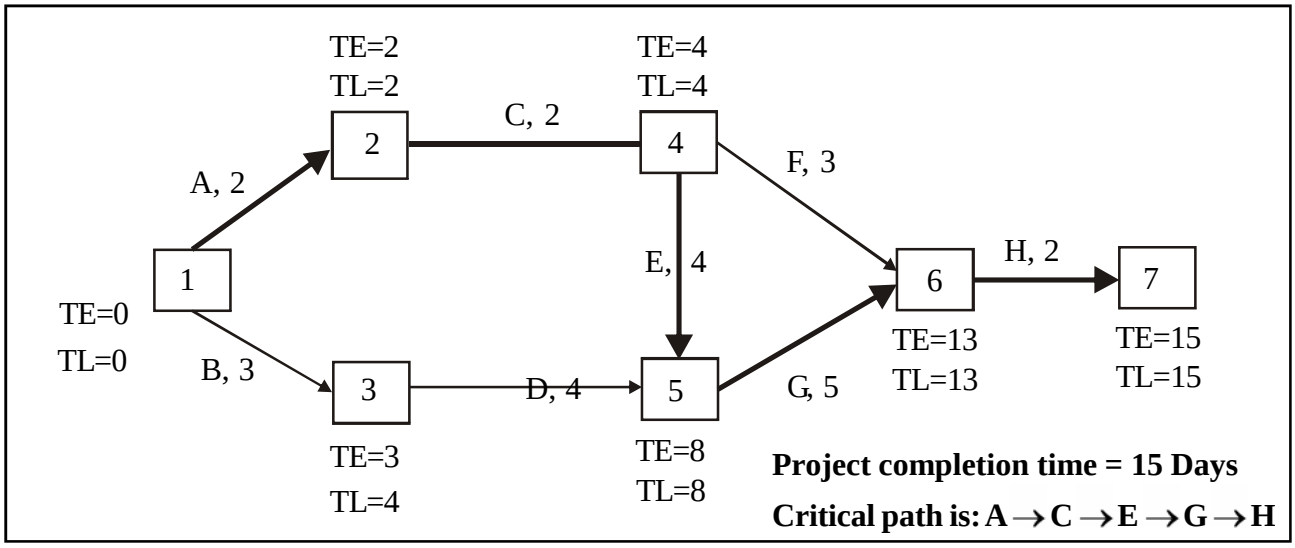


- Now we can next calculate latest completion time (TL) associated with each node. That is the latest time an activity can be completed without causing delay in the earliest completion date of the project. TL value of the sink node (last node) is equals its TE value.

  TL(Last node) = TE(Last node)

  TL(Node i) = TE(Last node) - max{t(p)} i ≠ last node

- Where t(p) denotes the sum of time durations for a path p from node i to last node, and maximum is taken overall such paths and subtracted from TE(Last node). Now calculate TL for the above diagram.



- After computed TE and TL values for each node, we can determine the critical path. Critical path is a path in which each node has its TE value equals to its TL value. (Or the path in which each node has slack time equals 0. Slack time = TL - TE). In above figure, the dark arrows show the critical path for given problem.

☛ **Note :** (Friends, આપણે ઉપરના example માં AoA (Activity On Arc) પદ્ધતિથી Activity Network દોર્યો. આપણે AoN પદ્ધતિથી પણ Activity Network દોરી આ જ example ગણી શકીએ છીએ. બંને પદ્ધતિઓમાં TE અને TL ની ગણતરી સમાન જ છે.)

4.4.4 GANTT chart :

- It was proposed by Henry Gantt in 1914.
- It is also called time line chart.
- It s mainly used to allocate resources to activities (Resource Planning).
- The resources allocated to activities include staff, hardware, and software etc.•

A Gantt chart is a special type of bar chart where each bar represents an activity.

- It is one of the most popular and useful ways of showing activities displayed against time.
- The bars are drawn along a time line. Activities against time are drawn in bar chart. The length of each bar is proportional to the duration of time planned for the corresponding activity.
- The slack time for a particular task is also shown in the bars.
- The chart is prepared by the project manager.

✢ **How to plan GANTT chart**

- Identify all the tasks.
- If possible, break down the tasks into smaller tasks.
- Determine the total estimated completion time for each task.
- Plot activities on GANTT chart. Draw milestones at applicable places.

✢ **Advantages**

- It is very simple to understand and easy to use.
- It is used in monitoring the progress of the project.
- GANTT chart mainly used for resource allocation.
- Useful for planning and guiding projects, understating critical paths & planning resources.

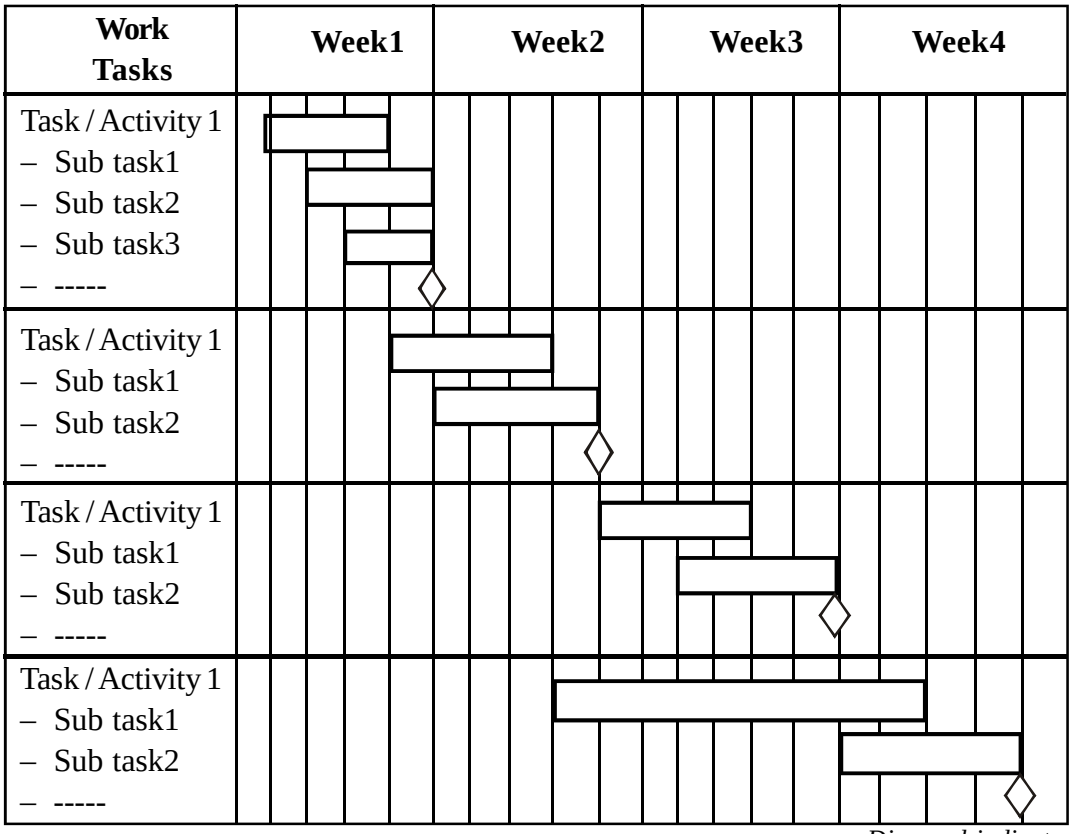
✢ **Disadvantages**

- It doesn't show interdependencies of activities.
- It doesn't show precedence relationship among activities.
- Not suitable for large projects.
- It can't calculate shortest time for any activity in the project.

✢ **Application**

- Used in industry to plan and schedule the activities and milestones.

✢ **Example (a simple frame work that shows the GANTT chart) :**



Diamond indicates milestones

Figure 3.1 GANTT chart example

4.4.5 Project Monitoring and Control (PMC) :

- Planning is one of the most important project activities. And without proper planning, project monitoring and control is not possible.
- **Monitoring** - collecting, recording, and reporting information concerning project performance that project manager and others wish to know.
- **Controlling** - it uses data from monitoring activity to bring actual performance from planned performance.
- Project monitoring and planning (PMC) activities take place in parallel with project execution, so the implementation should be corrective at appropriate level.
- The main purpose is to - *to ensure quality of final product.*

➠ **Why there is a need of PMC**

- Simply because we know that things don't always go according to plan.
- To detect and react appropriately to deviations and changes to plans.

➠ **What do we monitor and control**

- We need to monitor → men, machine, money, material, space, time, tasks, quality, performance and we need to control' time, cost and performance.
- PERT chart is mainly used for project monitoring and control.

➠ **Monitoring and controlling project work includes following activities :**

- Comparing the work that is occurring to the project management plan.
- Assessing work performance information to determine if any corrective or preventative actions are necessary.
- Analyzing, tracking, monitoring, and reporting on project risks.
- Providing status reports, accomplishments, and issue reports.
- Monitoring the implementation of approved changes.
- Do scope verification process, schedule control, quality control and cost control.
- Making sure that approved defect repairs have been made.
- Take corrective actions when needed.
- Monitoring and controlling outputs related to risk management include updating the risk register.
- Monitoring and controlling outputs related to communications management include performance reports, forecasts, and resolved issues.
- Techniques used for project monitoring and control activity are: Earned Value Analysis (EVA) and Critical ratio.
- A way of measuring overall performance (not individual task) is using an aggregate performance measure - Earned Value.
- **Earned Value Analysis** (EVA) is an industry standard method of measuring project's progress at any given point of time.
- Earned value of work performed (value completed) for those tasks in progress, found by multiplying the estimated percent physical completion of work for each task by the planned cost for those tasks. The result is amount that should be spent on the task so far. This can be compared with actual amount spent.
- Especially for large projects, it may be worthwhile calculating a set of critical ratios for all project activities

- The critical ratio is :

$$\frac{\text{Actual progress}}{\text{Scheduled progree}} * \frac{\text{Budgeted cost}}{\text{Actual cost}}$$

- If ratio is 1 everything is probably on target.
- The further away from 1 the ratio is, the more we may need to investigate.
- Continuous monitoring gives the project management team insight the health of the project, and identifies any areas that can require special attention.

4.5 RISK MANAGEMENT

- Tomorrow's problem is today's risk.
- Software risk is a problem that could cause some loss or threaten the success of software project, but which hasn't happened yet.
- This risk may affect negatively to the cost, schedule, technical success or quality of the project.
- Risk management is the process of identifying, addressing and eliminating the problems of risks before they can damage the project.

✢ **Objectives of the risk management**

- Identify potential problems and deal with them when they are easier to handle before they become critical.
- Focus on the project's objectives and consciously look for things that may affect project quality.
- Allow early identification of risks and provide management decisions to the solutions.
- Increase the chance of project success.

Risk management activities :

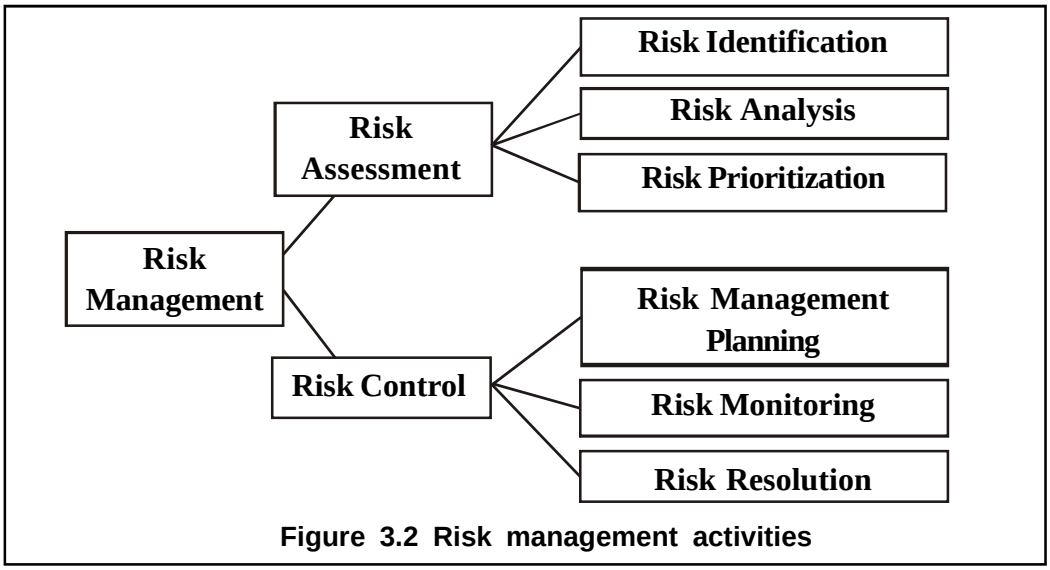


Figure 3.2 Risk management activities

4.5.1 Risk assessment :

- It is the process of examining a project and identifying areas of potential risk.
- It includes the following activities :
  - (i) Risk identification
  - (ii) Risk analysis
  - (iii) Risk prioritization

(i) Risk identification

- It is a systematic attempt to specify threats of the project plan.
- The purpose of risk identification is → to develop list of risk items also called risk statement.
- So some common risks areas are found and checklist is prepared.
- If we want to identify the important risks which may affect the project, it is necessary to categorize risks into different classes.
- The project manager can then examine which risks from each class are relevant to the project.
- There are three main categories of risks which can affect a software project :

1. Project risks

→ Project risks threaten the project. They concerned various forms of budgetary, schedule, personnel, resource, and customer-related problems.

→ An important project risk is schedule slippage. Because it is very difficult to monitor and control this risk.

2. Technical risks

→ Risks that threaten the quality of the product. These risks concern potential design, implementation, interfacing, testing, and maintenance problems.

→ It also covers the specification risks.

→ Most technical risks occur due to the development team's insufficient knowledge about the project.

3. Business risks

→ Risks that threaten the development (or client) organization.

→ This type of risks include risks of building an excellent product that no one wants, losing budgetary or personnel commitments, etc.

- The output of this activity is the list of risks.

(ii) Risk analysis

- When the risks have been identified, all risks are analyzed using different criteria.
- The purpose of this activity is → to examine how project outcomes might change with modification of risk input variables.
- The input of this activity is the list of risks developed in risk identification and output is the ranking of the risks and further analysis of description and probability of the risks.
- The main activities of this process are :

➠ Group similar risks - detect duplicates and group similar risk items.

➠ Determine risk drivers - determine risk parameters that affect identified risks.

➠ Determine source of risks - find out causes of risks.

➠ Estimate risk exposure - doing by measuring the probability and consequences (પરિણામ) of a risk.

➠ Evaluate against criteria - each risk item is evaluated using the predefined criteria, which are important for the specific risk.

(iii) Risk prioritization

- It helps the project focus on its most severe risks by assessing the risk exposure.
- This process can be done in a quantitative way, by estimating the probability (0.1 - 1.0) and relative loss, on a scale of 1 to 10.
- The higher the exposure, the more the risk should be tackled.
- Another way of handling risk is the risk avoidance (do not do risky things).

4.5.2 Risk control :

- Risk control is the process of managing risks to achieve the desired outcomes.
- Risk control activity includes the following :
  - (i) Risk management planning
  - (ii) Risk monitoring
  - (iii) Risk resolution

(i) Risk management planning

- It produces a plan for dealing with each significant risk.
- In involves identification of strategies to deal with risk. These strategies fall into three categories.
  - ➠ Risk avoidance - simply "don't do the risky things". We may avoid risks by not undertaking certain projects.
  - ➠ Risk avoidance attempts to reduce the probability of a risk.
  - ➠ Risk minimization (risk reduction) - to reduce the impact of the risk.
  - ➠ Risk contingency (અનિશ્ચિતતા) plan - which deals with a risk if it occurs.

(ii) Risk monitoring

- Risk monitoring is the continuous process of reassessing risks as the project proceeds and when the conditions change.
- Projects can be evaluated periodically to manage the risks and reduce their impact on the project.
- Each key risk should be discussed at management progress meetings.

(iii) Risk resolution

- When a risk occurred, it has to be solved. Risk resolution is the execution of the plans for dealing with each risk.
- The project manager has to decide the plan for resolving the risks.
- The input of this activity is $\rightarrow$ risk action plan and the outputs are :
  - ➠ Risk status
  - ➠ Acceptable risks
  - ➠ Reduced rework
  - ➠ Corrective actions
  - ➠ Problem prevention

Unit-5

Software Coding & Testing

5.1 CODE REVIEW (CODING CONCEPTS)

- In general, coding means set of guidelines for a specific programming language. Coding is what makes it possible for us to crate computer software, applications, websites etc.
- In the simple term, coding is → telling a computer what you want it to do, which involves typing in step-by-step commands for the computer to follow.
- Objective of coding → transform the design document into high level language code and unit test this code.
- Input to the coding phase → design document (module structure and data structure algorithms).
- Good software development organizations normally require their programmers to have some well-defined and standard style of coding that is coding standards.
- Coding standards, sometimes referred to as programming styles or coding convention.
- The purpose of coding standards :
  - ➠ It gives a uniform appearance to the codes written by different engineers.
  - ➠ It enhances code understanding.
  - ➠ It encourages good programming practices.
- Coding standards list several rules to be followed and coding guidelines provide general suggestions.
- Good software development organizations usually develop their own coding standards and guidelines depending on what best suits their organization and the type of products they develop.

5.1.1 Coding standards :

- Software coding standards are language-specific programming rules that greatly reduce the probability of introducing errors into your applications, regardless of which software development model (iterative, waterfall, extreme programming, and so on) is being used to create that application.
- The use of global should be limited.
- Contents of the headers preceding codes for different modules
  - ➠ Name of the module
  - ➠ Date in which module is created
  - ➠ Author's history
  - ➠ Global variables used in the module
- Naming conventions for global variables, local variables, and constant identifiers.
- Error returns conventions and exception handling mechanisms.

5.1.2 Coding guidelines :

Some of the coding guidelines are listed below.

- Do not use a coding style that is too clever or too difficult to understand.
- Do not use an identifier for multiple purposes.

- Each variable should be given a descriptive name indicating its purpose.
- The code should be well-documented.
- The length of any function should not exceed 10 source lines.
- Do not use 'goto' statements.

5.1.3 Characteristics of good coding :

- **Simplicity** - a code should be simple and easy to understand.
- **Readability** - a programmer should write simple code, which is easily readable by the user.
- **Good documentation** - code should be properly documented with header, comments, meaningful variable names etc. Software documentation can be classified into internal and external documentation.
- **Transportability** - a program should be easy to transport to different environments and computers.
- **Usability** - a code should have error recovery techniques that help the users for using the software or program.

5.1.4 Code review (નિરીક્ષણ, તપાસણી) :

- Conducting reviews is one of the important activities of system verification.
- Code review for a model is carried out after the module is successfully compiled and the all the syntax errors have been eliminated.
- According to IEEE standards, a code review is → "a process or meeting during which code is presented to project personnel, managers, users, customers or interested parties for comments or approval."
- Vulnerabilities in the code are found in this meeting and the quality of the code should be improved.
- Code reviews are extremely cost-effective strategies for reducing errors and to produce high quality code.

✢ **Advantages of code review**

- It improves code quality.
- It improves code readability.
- It improves the planning and estimation of project.
- Knowledge sharing is possible by code review.
- It helps us in improving domain expertise.

✢ **Classification of code review**

- Code reviews can be classified into → Formal review and Informal review.

➠ **Formal reviews** are conducted at the end of each life cycle phase. They are held when the author believes that the product is error free.

- Results of the formal reviews are documented.

➠ **Informal reviews** are conducted on as-needed basis. They may be held at any time, serving the purpose of brainstorming session and no agenda set.

- Results of the informal reviews are not documented.

✢ **Techniques of code review**

- Two main techniques for code review are carried out on the code of module :
  i. Code walkthrough
  ii. Code inspection

(i) Code walkthrough

- Code walk through is an informal code analysis technique proposed by Fagan.
- This technique can be used throughout the software lifecycle to assess and improve the quality of the software products.
- A Code Walkthrough is an informal meeting where the programmer leads the review team through his/her code and the reviewers try to identify faults.
- In the process of code walkthrough, after a module has been coded, successfully compiled and all syntax errors eliminated, a few members of the development team are given the code few days before the walk through meeting to read and understand code. Each member selects some test cases and simulates (અનાવટી) execution of the code by hand.
- Members note down their findings and discuss them in the meeting.
- Several guidelines are produced in the meetings and accepted as examples.
- Some of the prerequisite guidelines are the following.
  - → The team performing code walk through should not be either too big or too small. Ideally, it should consist of between three to seven members.
  - → Discussion should focus on discovery of errors and not on how to fix the discovered errors.

✢ **Advantages of code walkthrough**

- It is useful for the people if they are not from the software discipline; they are not used to or cannot easily understand software development process.
- It improves project team communication and morale of team members.
- It's an excellent educational medium for new team members.
- If it is done right, it can save time and improve quality over the project lifecycle.

✢ **Limitations of code walkthrough**

- The main disadvantage is that it takes more time.
- As this technique is informal, the documentation of this process is not done.

(ii) Code inspection

- Code inspection is a formal, efficient and economical method of finding faults in design and code proposed by Fagan [1976].
- The goal of code inspection is → to identify and remove bugs before testing the code and to discover the algorithmic and logical errors.
- During code inspection, the code is examined in the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walk through.
- Coding standards are also checked during code inspection.
- Good software companies list some common types of errors and these are discussed during code inspection to look out for possible errors.
- An inspection team consists of four persons who play the role of moderator, reader, recorder and author.
  - **Moderator :** he is a technically knowledgeable person. He leads the inspection process.
  - **Reader :** the reader takes the team through the code.

- **Recorder :** he notes each error on a standard form.
- **Authors :** he understands the errors found and try to solve unclear areas.

✢ **Following is a list of some general programming errors which can be checked during code inspection**

- Jumps into loop (in case of nested loop)
- Non terminating loops.
- Array includes out of bound.
- Improper storage allocation.
- Use of uninitialized variables.
- Mismatch between actual and formal parameters.
- Use of incorrect logical operations.
- Control flows and computational expressions.

✢ **Advantages of code inspection**

- List all potential design flows. That makes software code maintainable and less costly.
- A detailed error feedback is provided to individual parameters.
- It makes easier to change in the code.
- As this technique is formal, proper documentation of this method is done.

➥ *Code walkthrough is informal technique lead by an author while code inspection is formal technique lead by moderator.*

➥ *Code walkthrough and code inspection both are static testing techniques.*

5.2 SOFTWARE DOCUMENTATION

- Software documentation is an important aspect of both software projects and software engineering in general.
- It could be paper document (manuals, broacher etc) or an electronic document (text written inside code). Paper documents are called external documents and electronic documents are called internal documents.
- Software documentation can be defined as, an artefact (માનવ સર્જિત) whose purpose is to communicate information about the software system to which it belongs.
- Software documents work as an information repository for software engineers.
- It could be the part of the software (internal) and it can be available offline (external).
- When various kinds of software products are developed then not only the executable files and the source code are developed but also various kinds of documents such as users' manual, software requirements specification (SRS) documents, design documents, test documents, installation manual, etc are also developed as part of any software engineering process.
- Two main requirements for good documentation are → it is complete and up-to-date.

✢ **Advantages of good software documentation**

- Good documents enhance understandability and maintainability.
- Reduce effort and time for maintenance.
- Provides helps to users for effectively use of system.
- Good software documents helps in handling manpower turn over.
- Good software documents help the manager effectively track the progress of the system.
- Different types of software documents can be classified into two parts
  1. Internal documentation
  2. External documentation

1. Internal documentation

- It's a part of the source code itself.
- Internal documents are included in the syntax of the programming languages.
- Internal documentation is provided through appropriate module headers and comments embedded in the source code.
- The main objective of the internal documentation is to provide help to the user and the programmer to get a quick understanding of the program and the problem to modify the program as early as possible.
- It is also provided through the useful variable names, module and function headers, code indentation, code structuring, use of enumerated types and constant identifiers, use of user-defined data types, etc.
- Internal documents not only explain the programs, or program statements, but also help programmers to know before any action is taken for modification.
- Good internal documentation appropriately formulating by coding standards and coding guidelines.

2. External documentation

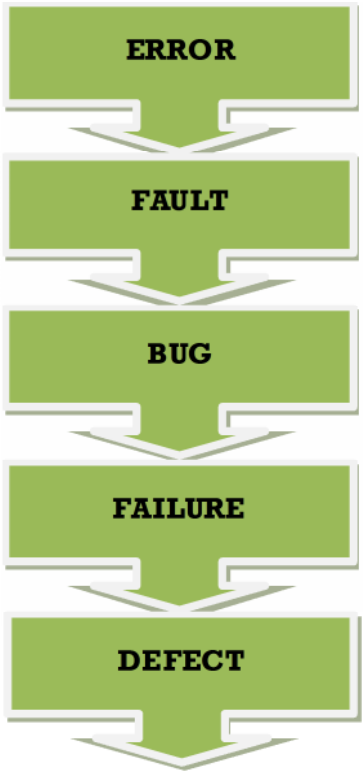
- These documents take place outside of the source code. It is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test documents, etc.
- External documents which focuses on general description of the software code and is not concerned with its detail.
- It includes information such as algorithms used in software code, dependencies of code in libraries, format of the output produced etc.
- External documents have two types : one for the users and one for those who wants to understand how the program works.
- It makes the user aware of the errors that occur while running the software code.
- All external documents are produced in orderly manner.

5.3 TESTING

- The basic goal of any software development is to produce software that has no errors or has few errors. As we know that faults can occur during any phase of software development cycle.
- Verification is performed at output of each phase, but some faults are likely to remain undetected and they can affect the whole software.

- Testing is relied on to detect these faults. Testing is itself an expensive activity.
- If program fails to behave as expected, it needs to be debugged and corrected. For that testing is done.
- Testing is the process of executing a program to locate an error.
- Testing a program consists of providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected.
- Aim of testing $\rightarrow$ to identify all defects existing in a software product.
- A good test case is one that has a high probability of finding undiscovered errors.

✢ **Some commonly used terms associated with testing are :**



- **Error :** Error is a kind of mistake. This may be a syntax error or misunderstanding of specifications. Sometimes it may be logical error.
- **Fault :** An error may lead to one or more faults. More precisely a fault is the representation of an error.
- **Bug :** A software bug is an error or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways.
- **Failure :** When software is unable to perform as per the requirements, it is called failure. Failure occurs when a fault executes. This is a demonstration of an error (or defect or bug).
- **Defect :** Defect is defined as the deviation from the actual and expected result of application or software

✢ **Some other testing terms are :**

- **Test case :** Test case is a set of conditions or variables under which a tester will determine whether a system under test works correctly or satisfy requirements.

  This is the triplet [I, S, O], where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system.

- **Test suite :** This is the set of all test cases with which a given software product is to be tested.
- Testing is four stage process :

  Unit testing → subsystem testing → system testing → acceptance testing.
- Initially all the modules are tested individually by their programmers, then interactions between these modules are tested (integration testing), then whole system as a single component is tested and finally the system is tested against users' data for final approval from the users.

✢ **Difference between verification and validation**

Verification	Validation
→ Verification is the process of confirming that software meets its specification.	→ Validation is the process of confirming that software meets customers' requirements.
→ Verification is the process of determining whether the output of one phase of software development confirms to that of its previous phase.	→ Validation is the process of determining whether a fully developed system confirms to its requirements specification.
→ Verification is concerned with phase containment of errors	→ Validation is concerned with final product be error free.
→ Verification → "are we doing right ?"	→ And Validation → "have we done right ?"
→ Verification comes before validation.	→ Validation comes before verification.
→ It is static testing.	→ It is dynamic testing.
→ Cost of verification is less.	→ Cost of validation is more.
→ Verification does not include the execution of code.	→ Validation includes the execution of code.

Both strategies are used to finds defects in software project, but in different way, Verification is used to identify the errors in requirement specifications & Validation is used to find the defects in the implemented software application.

✢ **Design of test cases**

- We must design an optimal test suite that is of reasonable size and can uncover as many errors existing in the system as possible.
- Testing a system using a large collection of test cases that are selected at random does not guarantee that all of the errors in the system will be uncovered.

Example :

```
int x, y;
take values of x and y
from user
if (x>y) then
        x is greater;
else
        y is greater;
```

- Invalid test suit :
  {(x=3, y=2), (x=8, y=4), (x=18, y=12)}
- Valid test suit:
  {(x=3, y=2), (x=4,y=5)}

- Systematic approach should be followed to design an optimal test suit. Each test case is design to detect different errors.
- There are mainly two approaches to systematically design test cases :

Black box testing	White box testing

- In the **black-box testing approach**, test cases are designed using only the functional specification of the software, i.e. without any knowledge of the internal structure of the software.
- So, black-box testing is known as functional testing.
- In the **white-box testing approach,** designing test cases requires thorough knowledge about the internal structure of software.
- So, the white-box testing is called structural testing.

✢ **Testing in the large vs. testing in the small**

- Software products are normally tested first at the individual component (or unit) level. This is referred to as testing in the small.
- After testing all the components individually, the components are slowly integrated and tested at each level of integration (integration testing). Finally, the fully integrated system is tested (called system testing).
- Integration and system testing are known as testing in the large.

➠ **Levels of testing**

- There are three levels of testing :
  - o **Unit testing :** it is first level of testing and individual module is tested.
  - o **Integration testing :** combine the modules and test their detailed structures.
  - o **System testing :** the whole system is tested ignoring the internal structure of the system.

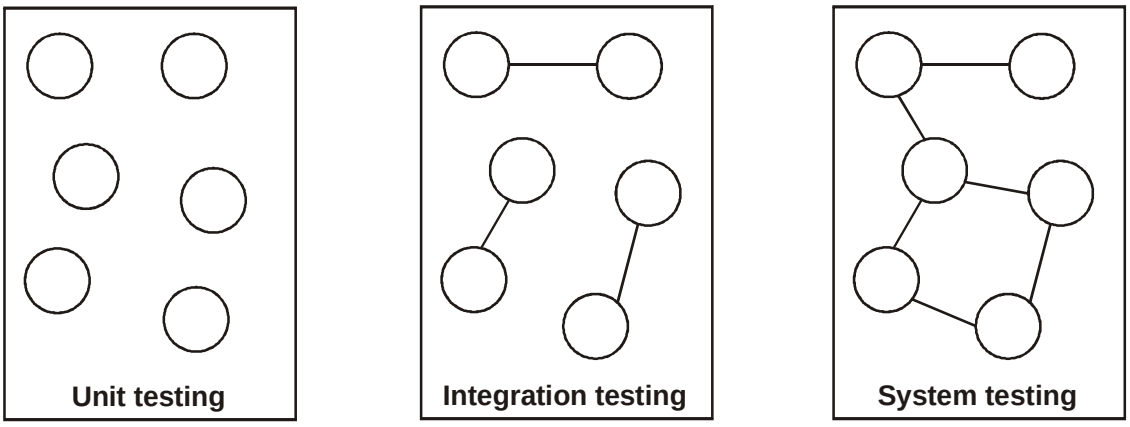


Figure 5.1 Level of testing

5.3.1 Unit testing :

- Unit testing is the first level of testing.
- It is the process of taking a program module and running it in isolation form from the rest of the product.
- Unit testing is undertaken after a module has been coded and successfully reviewed.
- Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation.

- This testing is a white box testing methodology.
- Purpose of unit testing is to validate that each unit of the software code performs as expected.
- Unit testing is done during development phase.
- A complete environment is needed to execute the unit testing on the module.
- Following steps are needed in order to test the module
  - o The procedures belonging to other driver module which contains non-local data structure.
  - o A procedure belonging to the module which is working as stub module.
  - o A procedure belonging to the module which is under test.
- The calling module and called module should be unit tested.
- An issue with the unit testing is that unit testing is done as a part, not the whole system. But in execution, one module may use other modules that have not been developed yet. For that dummy modules are needed.
- Due to this, unit testing often require driver and/or stub modules. The role of driver and stub modules is described in below figure.

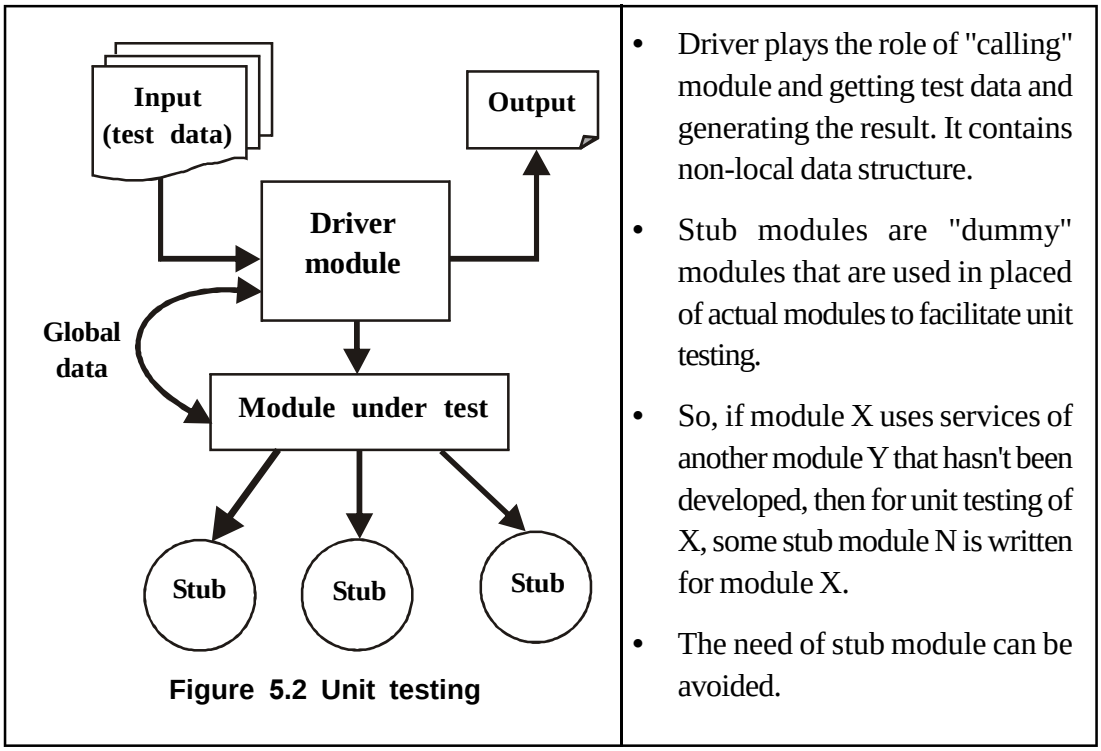


Figure 5.2 Unit testing

- Driver plays the role of "calling" module and getting test data and generating the result. It contains non-local data structure.
- Stub modules are "dummy" modules that are used in placed of actual modules to facilitate unit testing.
- So, if module X uses services of another module Y that hasn't been developed, then for unit testing of X, some stub module N is written for module X.
- The need of stub module can be avoided.

✢ **Advantages of unit testing :**

- Improve the quality of the code.
- Finds the software bugs early.
- One of the main benefits of unit testing is that it makes the coding process more agile. (ઝડપી, સરળ અને સુવ્યવસ્થિત)
- It simplifies integration testing.

5.3.2 Black box testing :

- This method is also called behavioural testing or functional testing.
- It is a testing method where test cases are designed using only the functional specification of the software, i.e. without any knowledge of the internal structure of the software.

• Functionality of the black box is understood completely in terms of its inputs and output.

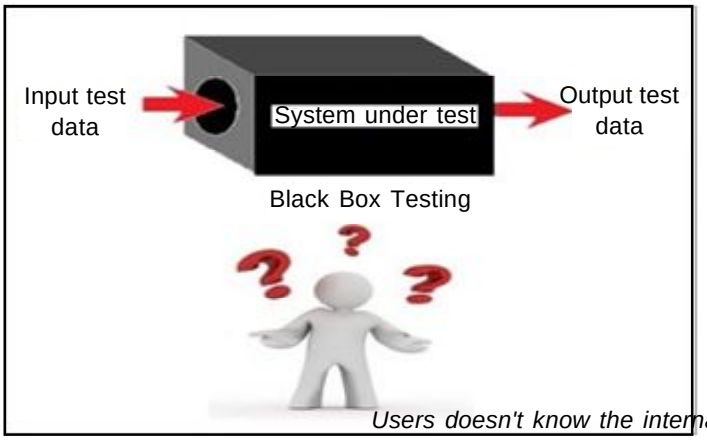
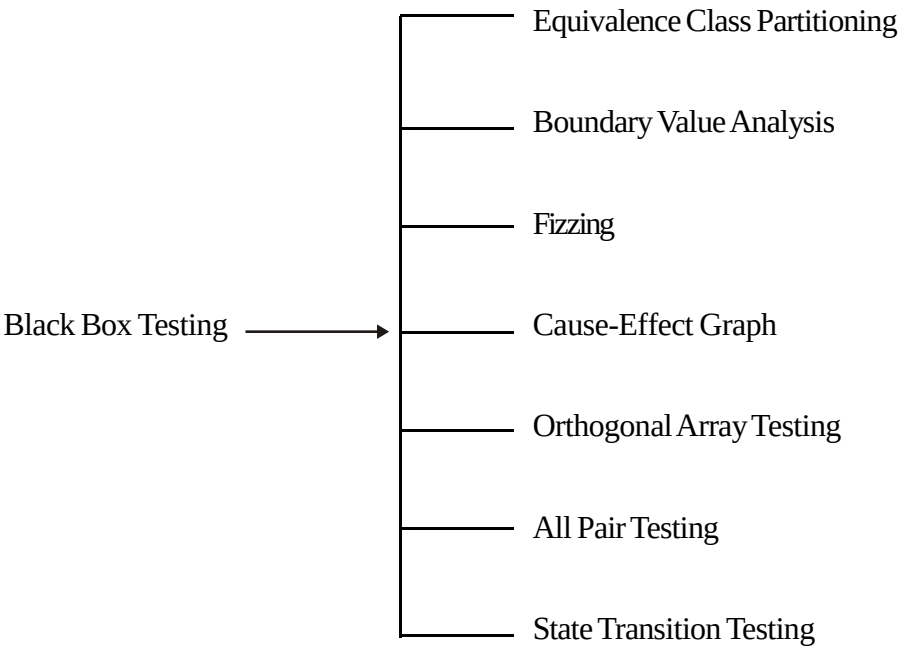


Figure 5.3 Black box testing

- Black box testing treats the software as a "Black Box" - without any knowledge of internal working and it only examines the fundamental aspects of the system. While performing black box test, a tester must know the system architecture and will not have access to the source code.
- The example of black box testing is search engine. You enter text that you want to search in the search bar, and results are returned to you. You don't know the specific process or algorithm of searching.
- There are number of techniques that can be used to design test cases in black box testing method, which are listed below.



Among all of the above, we will discuss only those techniques which are very successful in detecting errors.

✢ **Equivalence class partitioning**

- In it, the domain of input values (input test data) to a program is partitioned into a finite number of equivalence classes.
- So the behaviour of the program is similar for every input data belonging to the same equivalence class.

- The main idea behind defining the equivalence classes is that if one test case in a class detects an error all other test cases in the class would be expected to find same error.
- To implement this technique two steps are required.
  (i) The equivalence classes are identified by taking each input condition and partitioning in into valid and invalid classes. For example, if an input condition specifies a range of values from 1 to 100, we identify one valid equivalence class (1 to 100); and two invalid equivalence classes (< 1) and (>100).
  (ii) Generate test cases using the equivalence classes identified in the previous step. Now perform testing on both valid and invalid equivalence classes.

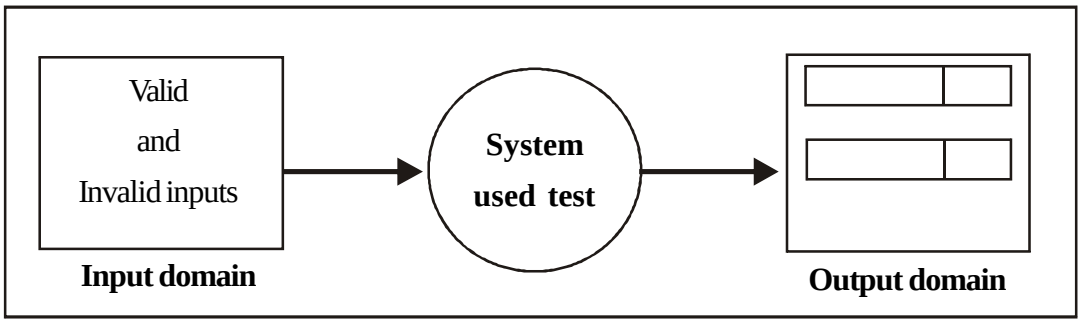


Figure 5.4 Equivalence partitioning

- The aim is to choose at least one element from each equivalence class.
- The following are some general guidelines for designing the equivalence classes :
  - If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.
  - If the input data assumes values from a set of discrete members of some domain, then one equivalence class for valid input values and another equivalence class for invalid input values should be defined.
  - If the input value is Boolean, then one valid and one invalid equivalence classes should be defined.
- For example, for a program that supposes to accept any number between 1 and 99, there are at least four equivalence classes from input side. Like :
  i. Any number between 1 and 99 is valid input.
  ii. Any number less than 1 (includes 0 and all negative numbers).
  iii. Any number greater than 99.
  iv. If it is not a number, it should not be accepted.

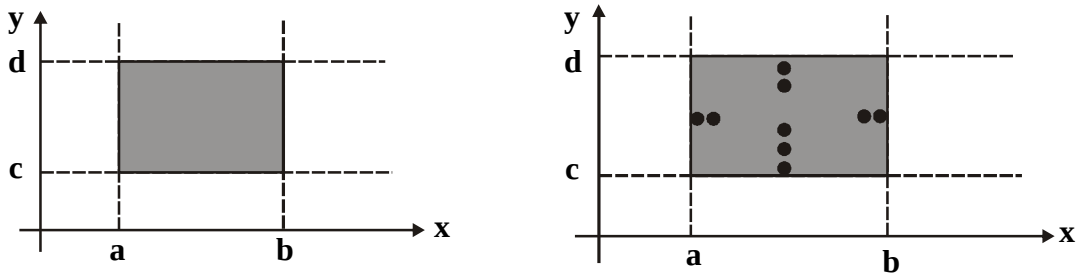
✢ **Boundary value analysis (BVA)**

- Generally errors are occurred at boundary of domains rather than centre of domain.
- This is because test cases closer to boundary have more chance to detect errors.
- For this reason, boundary value analysis technique has been developed.
- In this technique, selection of test cases performed at the edges of the class. Suppose we have an input variable x with a range from 1 to 100. The boundary values are : 1, 2, 99, 100.
- Consider a program with two input variables x and y. These input variables have specified boundaries as :

$$a \leq x \leq b$$

$$c \leq y \leq d$$

- Both the inputs x and y are bounded by two intervals [a,b] and [c,d] respectively. For input x, we may design test cases with values a and b, just above a and also just below b. for input y, we may have values c and d, just above c and just below d. These test cases have more chance to detect an error. Like :



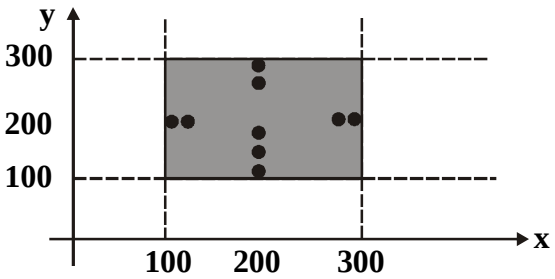
"Each dot represents a test case and inner rectangle is the input domain."

- Basic idea of BVA is to use input variables values at their minimum, just above minimum, a nominal value, just below maximum and at their maximum.
- BVA test cases are obtained by holding the values of all but one variable at their nominal value.

➠ **Example :**

- The test cases for BVA in a program with two input variables x and y that may have any value from 100 to 300 are:

  (200,100), (200,101), (200,200), (200,299), (200,300), (100,200), (101,200), (299,200) and (300,200). (given in below figure).

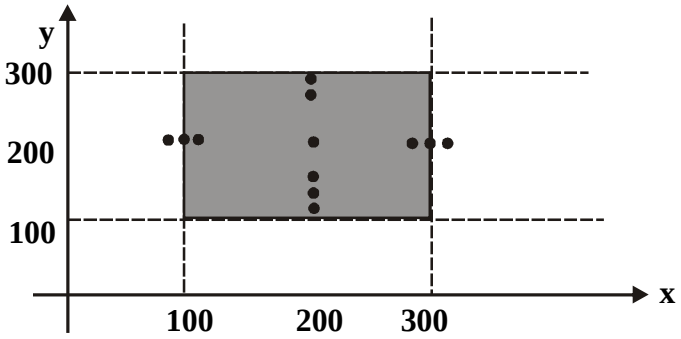


Input domain

"Thus for a program of n variables, BVA yields 4n + 1 test cases."

✢ **Robust testing**

- It is an extension of boundary value analysis.
- In this type of testing, the extreme values are exceeded with a slightly greater than the maximum and a value slightly less than the minimum. This is shown in below figure.



- For this technique, if a program of n variables, then BVA yields 6n + 1 test cases.

  Test cases are : (200,99), (200,100), (200,101), (200,200), (200,299), (200,300), (200,301), (99,200), (100,200), (101,200), (299,200), (300,200), (301,200).

✢ **Advantages of black box testing**

- It is Efficient for large code segment.
- Tester doesn't need to know the internal structure of the system.
- Tester perception is very simple.
- Programmer and tester are independent of each other.
- Quicker test case development.

✢ **Disadvantages of black box testing**

- It is inefficient testing.
- Without clear specification test cases are difficult to design.
- Only a selected number of test scenarios are actually performed. As a result, there is only limited coverage.

"Black box testing is also known as close box testing and opaque testing."

5.3.3 White box testing :

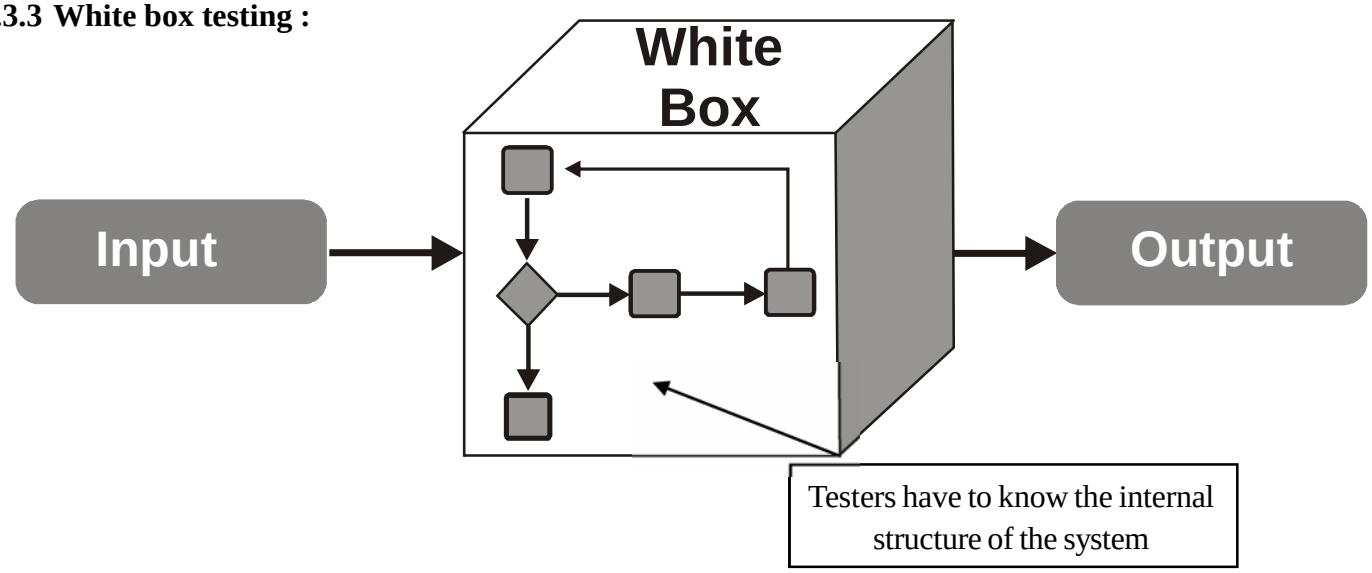
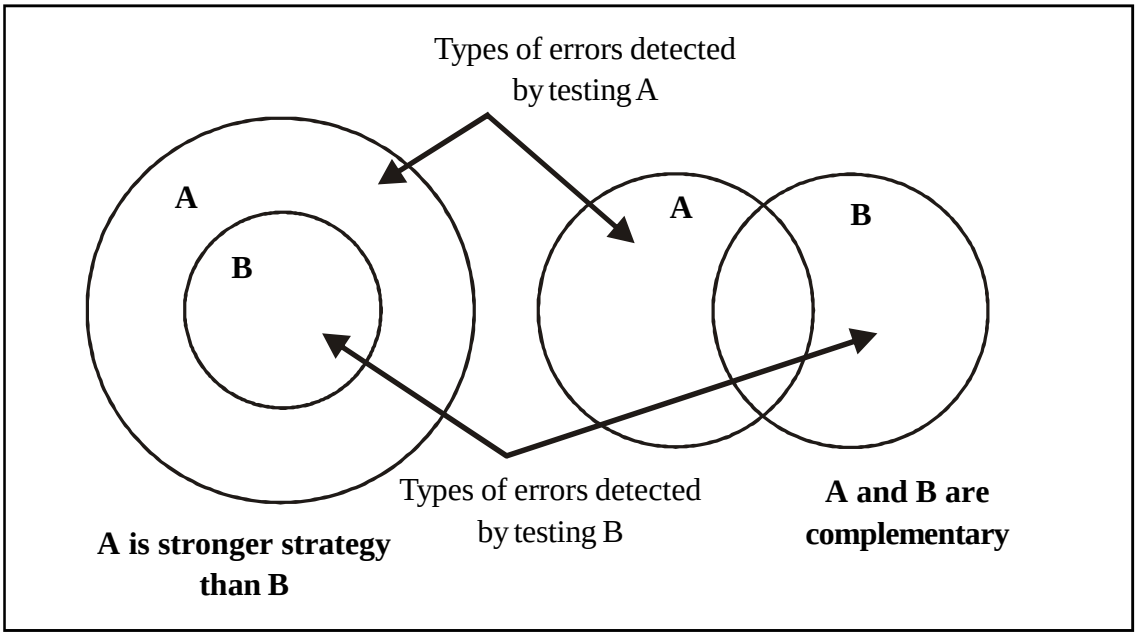
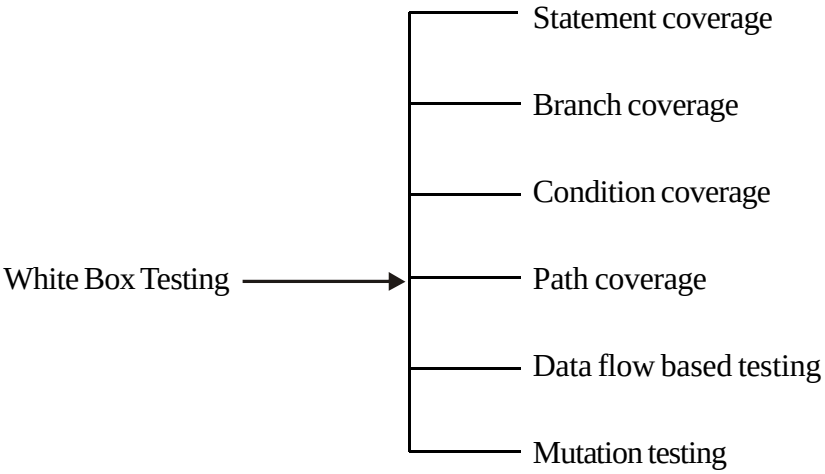


Figure 5.5 White box testing

- This method is concerned with testing the implementation of the program.
- The aim of this testing is to investigate the internal logic and structure of the code. That is why white box testing is also called structural testing.
- In white box testing it is necessary for a tester to have full knowledge of source code.
- Some of the synonyms of white box testing are glass box testing, clear box testing, open box testing, transparent box testing, structural testing, logic driven testing and design based testing.
- There are six basic types of testing : unit, integration, function/system, acceptance, regression, and beta. White-box testing is used for three of these six types : Unit, integration and regression testing.
- There are many white box strategies available, in which one is stronger than another. When two testing strategies detect errors that are different, then they are called complementary.
- The concepts of stronger and complementary testing are schematically illustrated in given figure.



- If we test program while it is running, it is called dynamic white box testing, and if we test the program without running it, only by examining and reviewing it then it is called static white box testing.
- Test cases are designed in this method are based on program structure or logic. Example of white box testing is circuit testing. As in electric circuit testing, the internal structure is checked.
- Test cases generated using white box testing can :
  - Guarantee that all independent paths within a module have been exercised at least once.
  - Exercise all decisions whether they are true or false.
  - Exercise internal data structure of the program.
- The different methods of white box testing are illustrated in below figure.



✢ **Statement coverage :**

- The statement coverage is also known as line coverage or segment coverage.
- The principal idea behind the statement coverage strategy is that unless a statement is executed, it is very hard to determine if an error exists in that statement.
- It aims to design test cases so that every statement in a program is executed at least once.

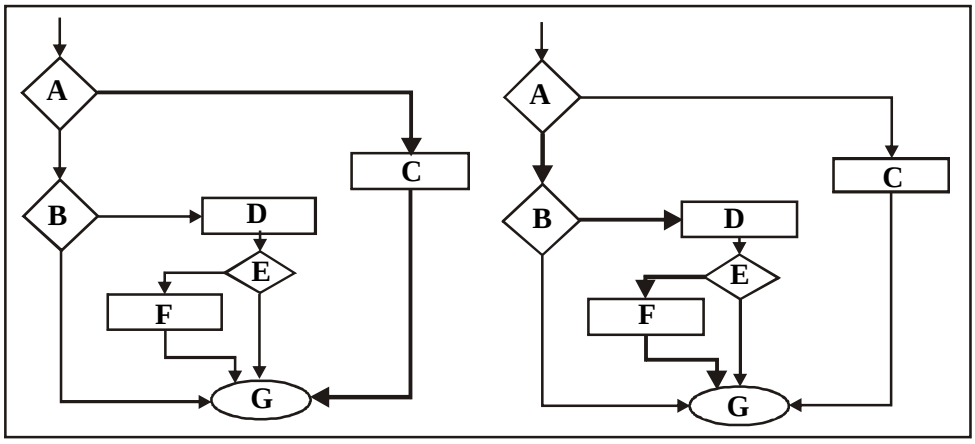
- Statement coverage is also known as node testing.
- However, executing some statement once and observing that it behaves properly for that input value is no guarantee that it will behave correctly for all other input values.

Example :

```
int x, y;
if x > y
        printf("x is greater");
else
        printf("y is greater");
```

For this code test case should be :

{(5,4), (4,5)}



In above figure, we can see that all the nodes (A to G) have been tested by designing test cases in two different ways. All nodes are covered, that's why this strategy is called node testing.

✢ **Branch coverage :**

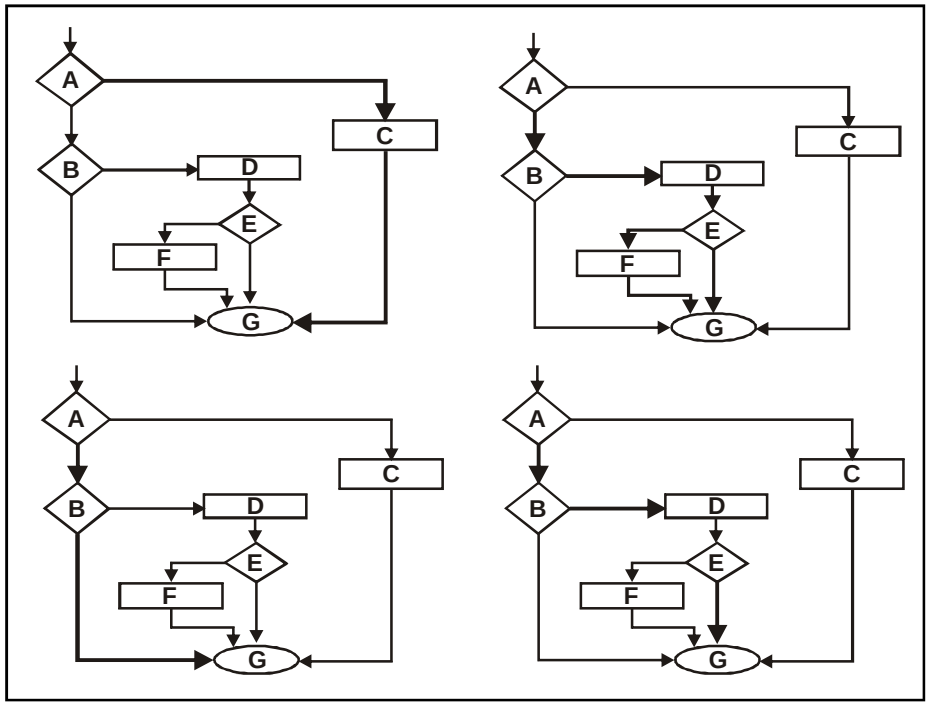
- In it, test cases are designed to make each branch condition to assume true and false values in turn.
- Branch testing is also known as edge testing as in it, each edge of a program's control flow graph is traversed at least once.
- Branch testing guarantees statement coverage, so it is stronger testing strategy than statement coverage.
- We can take above example for branch testing as well.

Example :

```
int compute_gcd(x, y)
int x, y;
{
1   while (x! = y){
2       if (x>y) then
3           x= x - y;
4       else y= y - x;
5   }
6    return x;
    }
```

For this code test case should be:

{(x=3,y=3), (x=4,y=3), (x=3,y=4)}



In above figure, we can see that test cases are designed in a way that every edge has been covered at least one. All the edges are covered that's why this testing strategy is called edge testing. At a particular time the edge which is examined is shown using bold arrow.

☛ **Note :** *Using statement and branch coverage user generally attains 80-90% code coverage, which is sufficient.*

✢ **Condition coverage :**

- In this method test cases are designed to make each component of a composite conditional expression to assume both true and false value.
- For example, consider the following code :

  1. READ X, Y
  2. IF(X == 0 || Y == 0)
  3. PRINT '0'

  In this example, there are two conditions ' X = 0 and Y = 0. Now test cases should be designed in a way that all conditions get TRUE and FALSE.

  Test case 1: (X=0, Y=5)

  Test case 2: (X=5, Y=0)
- Condition testing is stronger than branch testing.
- For a composite conditional expression, if n components are there, for condition coverage ' 2n test cases are required.
- In condition coverage, test cases are increasing exponentially with the number of components. Therefore, a condition coverage based testing technique is practical only if n (the number of conditions) is small.

✢ **Path coverage :**

- Path testing is used for module or unit testing.
- It requires complete knowledge of the program structure.
- The effectiveness of path testing deteriorates (ક્ષમતા ઘટવી) as the size of the software under test increases.

- It is not useful for system testing.
- This type of testing involves :
  - ➠ Generating a set of paths that will cover every branch in the program.
  - ➠ Finding a set of test cases that will execute every path in this set of program path.

✢ **Data flow based testing :**

- Data flow-based testing method selects test paths of a program according to the locations of the definitions and uses of different variables in a program.
- For a statement numbered S, let

  DEF(S) = {X/statement S contains a definition of X}, and

  USES(S) = {X/statement S contains a use of X}
- Ex For the statement S:**a=b+c;, DEF(S) = {a}. USES(S) = {b,c}**

✢ **Mutation testing :**

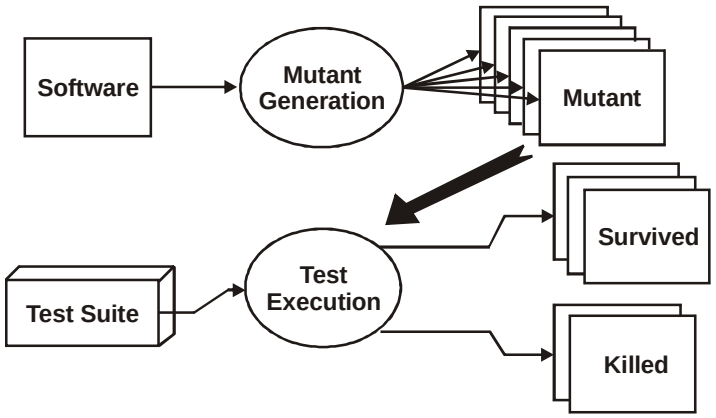


Figure 4.6 Mutation Testing

- Mutation testing is a type of white box testing which is mainly used for unit testing.
- In it, software is first tested by initial test suit built from different white box techniques, and then mutation testing is taken up.
- Main idea behind this technique is → to make few changes to program at a time. In this type of software testing where we mutate (change) certain statements in the source code and check if the test cases are able to find the errors.
- Each time a program is changed, it is called mutated program and effected change is called a mutant.
- After comparison of the results of original and mutant programs, if the original program and mutant programs generate the same output, then that the mutant is killed by the test case. Hence the test case is good enough to detect the change between the original and the mutant program. If the original program and mutant program generate different output, Mutant is kept alive. In such cases, more effective test cases need to be created that kill all mutants.
- Major disadvantage of mutation testing is → it is very expensive to compute as large number of mutant can be generated.
- It is not suitable for manual testing.

➠ **Advantages of white box testing methods**

- Reveal (બતાવવું, ઉજાગર કરવું) hidden errors in the code.
- White box testing is very thorough as the entire code and structures are tested.

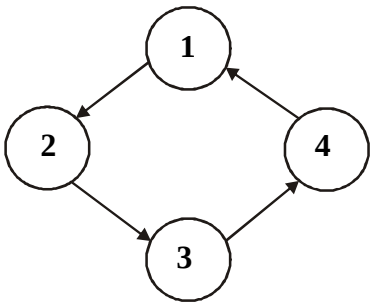
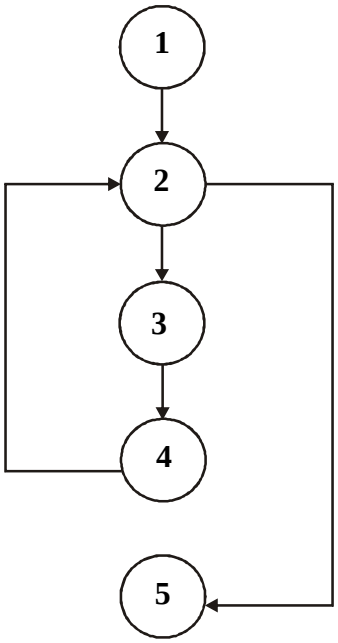
- Testing can start early in SDLC even if GUI is not available.
- Out product will be qualitative if white box testing is successful.

➠ **Disadvantages of white box testing methods**

- White box testing can be quite complex and expensive.
- It is time consuming as it takes more time to test fully.
- It requires professional resources.
- In-depth knowledge about the programming language is necessary to perform white box testing.

5.3.4 Control Flow Graph (CFG) :

- A control flow graph describes the sequence in which the different instructions of a program get executed. In other words, a control flow graph describes how the control flows through the program.
- In order to draw the control flow graph of a program, all the statements of a program must be numbered first. The different numbered statements serve as nodes of the control flow graph.
- An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node.
- The CFG for any program can be easily drawn by knowing how to represent the sequence, selection, and iteration type of statements in the CFG.
- Below figure summarizes how the CFG for these three types of statements can be drawn.

Sequence	Selection	Iteration
1. a=5 2. b=a*5	1. if (a>b) 2. print(a is greater) 3. else print(b is greater) 4. print(a+b)	1. a=5 2. while(a>0) { 3. print(a) 4. a=a-1 } 5. b=a;
		

- Using these basic concepts, the CFG of Euclid's GCD computation algorithm can be drawn as shown.

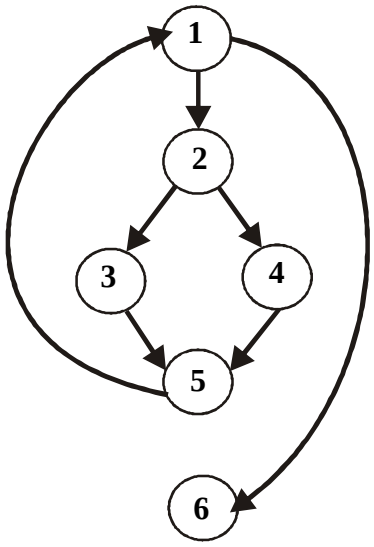
GCD algorithm	CFG for GCD algorithm
1. while (x != y) 2. { if (x > y) then 3. x = x - y 4. else y = y - x 5. } 6. return (x)	

Figure 4.7 CFG

✢ **Path**

- A path through a program is a node and edge sequence from the starting node to a terminal node of the control flow graph of a program. There can be more than one terminal node in a program.
- Path coverage requires the coverage of linearly independent path.

✢ **Linearly independent path**

- Linearly independent path is defined as a path that has at least one new edge which has not been traversed before in any other paths.
- If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent.
- Sub path is not considered as a linearly independent path.

✢ **Cyclomatic complexity**

- McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program.
- It is very simple to compute.

☛ Cyclomatic complexity is software metric used to measure the complexity of a program. This metric measures independent paths through program source code.

- Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program.
- It is practical way to determine maximum number of independent path in the program.

There are three different ways to compute the cyclomatic complexity.

➠ **Method 1 :**

- Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:

  V(G) = E - N + 2

  (where N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.)
- In above figure 4.6cyclomatic complexity according to method 1 is:

  V(G) = 7 - 6 + 2 = 3

➠ **Method 2 :**

- Another way of computing the cyclomatic complexity V(G) is

  V(G) = Total number of bounded areas + 1

  any region enclosed by nodes and edges can be called as a bounded area.
- In above figure 4.6, we can see two bounded area. So cyclomatic complexity is :

  V(G) = 2 + 1 = 3

➠ **Method 3 :**

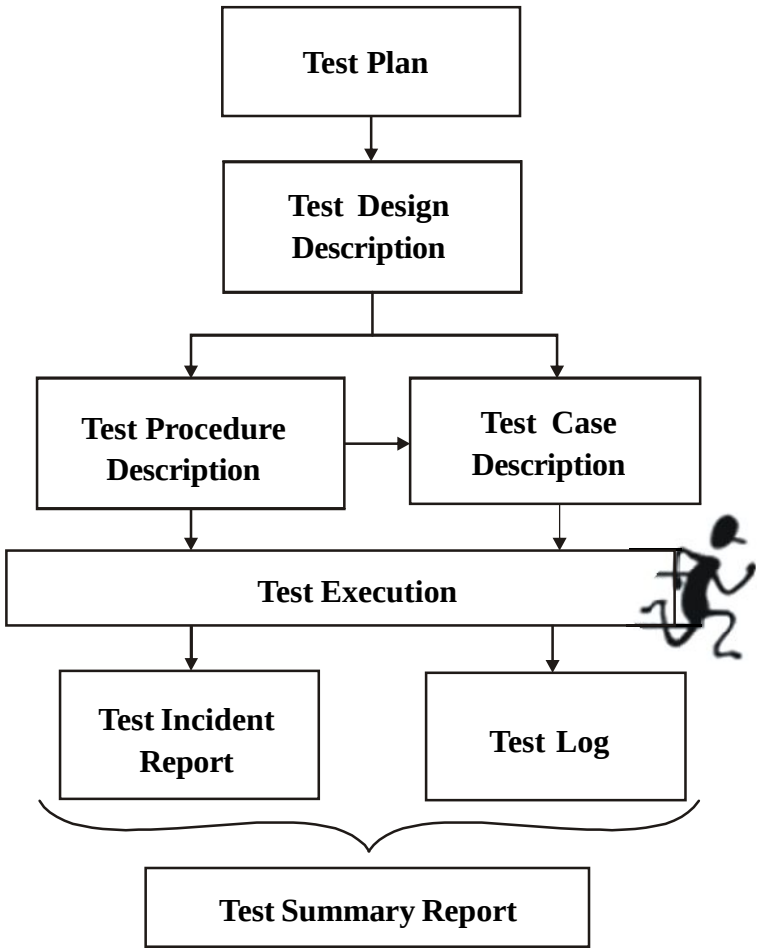
- The cyclomatic complexity of a program can also be easily computed by computing the number of decision statements of the program. If N is the number of decision statement of a program, then the McCabe's metric is equal to N+1. In our example, we can clearly see there are two decision statements available (while loop and if statement). So the cyclomatic complexity is : 2 + 1 = 3.

✢ **Difference between Black box testing and White box testing :**

Black box testing	White box testing
• Synonyms of black box testing are functional testing, close box testing, data driven testing and opaque testing.	• Synonyms of white box testing are structural testing, glass box testing, clear box testing, open box testing, logic driven testing.
• No need to know internal structure of the system.	• Internal structure of the system must be known.
• It is concerned with results.	• It is concerned with details and internal workings of the system.
• Performed by end users and also by testers and developers.	• Normally done by testers and developers.
• Granularity is low.	• Granularity is low.
• It is least exhaustive and time consuming.	• Potentially most exhaustive and time consuming.
• Not suited for algorithm testing.	• It is suited for algorithm testing.
• Example : search engine.	• Example : electrical circuit testing.

5.4 TEST DOCUMENTATION

- The documentation which is generated towards the end of testing or during the testing, it is the test summary reports or test documentation.
- It provides summary of test suits which has been applied to the system.
- It specify how many test suits are successful, how many are unsuccessful and what is the degree of successful and unsuccessful.
- Test documentation should be followed the IEEE standards 829.
- Test documentation should includes : scope, approach, resources and schedule of the testing activity to identify the items being tested, the features to be tested, the testing tasks to be performed, responsible personnel and the associated risks.



- A test design specification/description → to identify the features to be tested and associated tests.
- A test case specification/description → to define test cases identified by test design specifications.
- A test procedure specification → to specify the steps for executing a set of test cases.
- A test item transmittal report → to identify the test items being transmitted for testing.
- A test log' to document the generated results.
- A test incident report → to document occurred events during testing process.
- A test summary report → to summarize the results of the testing activities and to provide evaluation of these results.