LEARNING MATERIAL

ON

SOFTWARE ENGINEEARING

(5TH SEMESTER)

DEPARTMENT OF INFORMATION TECHNOLOGY

Prepared by
Smt. Madhusmita Dalai
Lect. IT
Govt.Polytechnic,Bhubaneswar

Introduction to Software Engineering

Software is a program or set of programs containing instructions that provide desired functionality. And Engineering is the process of designing and building something that serves a particular purpose and finds a cost-effective solution to problems.

Software Engineering *is a systematic, disciplined, quantifiable study and approach to the design, development, operation, and maintenance of a software system.*

Objectives of Software Engineering:

1. **Maintainability –**
   It should be feasible for the software to evolve to meet changing requirements.
2. **Efficiency –**
   The software should not make wasteful use of computing devices such as memory, processor cycles, etc.
3. **Correctness –**
   A software product is correct if the different requirements as specified in the SRS document have been correctly implemented.
4. **Reusability –**
   A software product has good reusability if the different modules of the product can easily be reused to develop new products.
5. **Testability –**
   Here software facilitates both the establishment of test criteria and the evaluation of the software with respect to those criteria.
6. **Reliability –**
   It is an attribute of software quality. The extent to which a program can be expected to perform its desired function, over an arbitrary time period.
7. **Portability –**
   In this case, the software can be transferred from one computer system or environment to another.
8. **Adaptability –**
   In this case, the software allows differing system constraints and the user needs to be satisfied by making changes to the software.
9. **Interoperability** – Capability of 2 or more functional units to process data cooperatively.

Program vs Software Product:

1. A program is a set of instructions that are given to a computer in order to achieve a specific task whereas software is when a program is made available for commercial business and is properly documented along with its licensing. Software=Program+documentation+licensing.

2. A program is one of the stages involved in the development of the software, whereas a software development usually follows a life cycle, which involves the feasibility study of the project, requirement gathering, development of a prototype, system design, coding, and testing.

Emergence of Software Engineering

Software engineering discipline is the result of advancement in the field of technology. In this section, we will discuss various innovations and technologies that led to the emergence of software engineering discipline.

- Early Computer Programming
- High Level Language Programming
- Control Flow Based Design
- Data-Flow Oriented Design Object Oriented Design

Early Computer Programming

As we know that in the early 1950s, computers were slow and expensive. Though the programs at that time were very small in size, these computers took considerable time to process them. They relied on assembly language which was specific to computer architecture. Thus, developing a program required lot of effort. Every programmer used his own style to develop the programs.

High Level Language Programming

With the introduction of semiconductor technology, the computers became smaller, faster, cheaper, and reliable than their predecessors. One of the major developments includes the progress from assembly language to high-level languages. Early high level programming languages such as COBOL and FORTRAN came into existence. As a result, the programming became easier and thus, increased the productivity of the programmers. However, still the programs were limited in size and the programmers developed programs using their own style and experience.

Control Flow Based Design

With the advent of powerful machines and high level languages, the usage of computers grew rapidly: In addition, the nature of programs also changed from simple to complex. The increased size and the complexity could not be managed by individual style. It was analyzed that clarity of control flow (the sequence in which the program's instructions are executed) is of great importance. To help the programmer to design programs having good control flow structure, **flowcharting technique** was developed. In flowcharting technique, the algorithm is represented using flowcharts. A **flowchart** is a graphical representation that depicts the sequence of operations to be carried out to solve a given problem.

Data-Flow Oriented Design

With the introduction of very Large Scale Integrated circuits (VLSI), the computers became more powerful and faster. As a result, various significant developments like networking and GUIs came into being. Clearly, the complexity of software could not be dealt using control flow based design. Thus, a new technique, namely, **data-floworiented** technique came into existence. In this technique, the flow of data through business functions or processes is represented using **Data-flow Diagram (DFD). IEEE** defines a data-flow diagram (also known as **bubble chart** and **work-flow diagram)** as 'a diagram that depicts data sources, data sinks, data storage, and processes performed on data as nodes, and logical flow of data as links between the nodes.'

Object Oriented Design

Object-oriented design technique has revolutionized the process of software development. It not only includes the best features of structured programming but also some new and powerful features such as encapsulation, abstraction, inheritance, and polymorphism. These new features have tremendously helped in the development of well-designed and high-quality software. Object-oriented techniques are widely used these days as they allow reusability of the code. They lead to faster software development and high-quality programs. Moreover, they are easier to adapt and scale, that is, large systems can be created by assembling reusable subsystems.

Computer system engineering

Computer systems engineering is a discipline that embodies the science and technology of design, construction, implementation, and maintenance of software and hardware components of modern computing systems, computer-controlled equipment, and networks of intelligent devices

Software Development Life Cycle (SDLC)

A software life cycle model (also termed process model) is a pictorial and diagrammatic representation of the software life cycle. A life cycle model represents all the methods required to make a software product transit through its life cycle stages. It also captures the structure in which these methods are to be undertaken.

In life cycle model maps the various activities performed on a software product from its inception to retirement. Different life cycle models may plan the necessary development activities to phases in different ways. Thus, no element which life cycle model is followed, the essential activities are contained in all life cycle models though the action may be carried out in distinct orders in different life cycle models. During any life cycle stage, more than one activity may also be carried out

SDLC Cycle

SDLC Cycle represents the process of developing software. SDLC framework includes the following steps:

The stages of SDLC are as follows:

Stage1: Planning and requirement analysis

Requirement Analysis is the most important and necessary stage in SDLC.

The senior members of the team perform it with inputs from all the stakeholders and domain experts or SMEs in the industry.

Stage2: Defining Requirements

Once the requirement analysis is done, the next stage is to certainly represent and document the software requirements and get them accepted from the project stakeholders.

This is accomplished through "SRS"- Software Requirement Specification document which contains all the product requirements to be constructed and developed during the project life cycle.

Stage3: Designing the Software The next phase is about to bring down all the knowledge of requirements, analysis, and design of the software project. This phase is the product of the last two, like inputs from the customer and requirement gathering.

Stage4: Developing the project

In this phase of SDLC, the actual development begins, and the programming is built. The implementation of design begins concerning writing code. Developers have to follow the coding guidelines described by their management and programming tools like compilers, interpreters, debuggers, etc. are used to develop and implement the code.

Stage5: Testing

After the code is generated, it is tested against the requirements to make sure that the products are solving the needs addressed and gathered during the requirements stage.

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

Stage6: Deployment

Once the software is certified, and no bugs or errors are stated, then it is deployed.

Then based on the assessment, the software may be released as it is or with suggested enhancement in the object segment.

After the software is deployed, then its maintenance begins.

Stage7: Maintenance

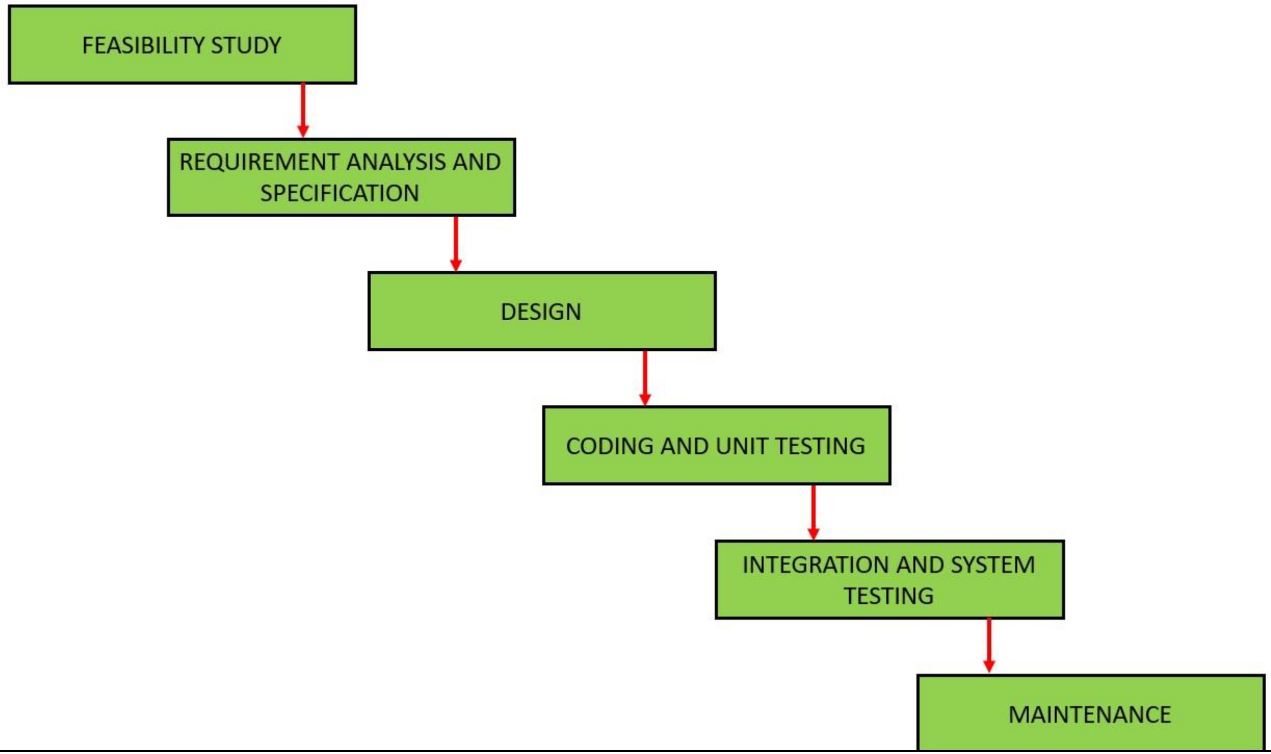
Once when the client starts using the developed systems, then the real issues come up and requirements to be solved from time to time.

This procedure where the care is taken for the developed product is known as maintenance.

Classical Waterfall

The classical waterfall model is the basic **software development life cycle** model. It is very simple but idealistic. Earlier this model was very popular but nowadays itx is not used. But it is very important because all the other software development life cycle models are based on the classical waterfall model.

The classical waterfall model divides the life cycle into a set of phases. This model considers that one phase can be started after the completion of the previous phase. That is the output of one phase will be the input to the next phase. Thus the development process can be considered as a sequential flow in the waterfall. Here the phases do not overlap with each other. The different sequential phases of the classical waterfall model are shown in the below figure:


FEASIBILITY STUDY
REQUIREMENT ANALYSIS AND SPECIFICATION
DESIGN
CODING AND UNIT TESTING
INTEGRATION AND SYSTEM TESTING
MAINTENANCE

1. **Feasibility Study**: The main goal of this phase is to determine whether it would be financially and technically feasible to develop the software.
   The feasibility study involves understanding the problem and then determining the various possible strategies to solve the problem. These different identified solutions are analyzed based on their benefits and drawbacks, The best solution is chosen and all the other phases are carried out as per this solution strategy.

2. **Requirements analysis and specification**: The aim of the requirement analysis and specification phase is to understand the exact requirements of the customer and document them properly. This phase consists of two different activities.
   - **Requirement gathering and analysis:** Firstly all the requirements regarding the software are gathered from the customer and then the gathered requirements are analyzed. The goal of the analysis part is to remove incompleteness (an incomplete requirement is one in which some parts of the actual requirements have been omitted) and inconsistencies (an inconsistent requirement is one in which some part of the requirement contradicts some other part).
   - **Requirement specification:** These analyzed requirements are documented in a software requirement specification (SRS) document. SRS document serves as a contract between the development team and customers. Any future dispute between the customers and the developers can be settled by examining the SRS document.
3. **Design**: The goal of this phase is to convert the requirements acquired in the SRS into a format that can be coded in a programming language. It includes high-level and detailed design as well as the overall software architecture. A Software Design Document is used to document all of this effort (SDD)

4. **Coding and Unit testing**: In the coding phase software design is translated into source code using any suitable programming language. Thus each designed module is coded. The aim of the unit testing phase is to check whether each module is working properly or not.

5. **Integration and System testing**: Integration of different modules are undertaken soon after they have been coded and unit tested. Integration of various modules is carried out incrementally over a number of steps. During each integration step, previously planned modules are added to the partially integrated system and the resultant system is tested. Finally, after all the modules have been successfully integrated and tested, the full working system is obtained and system testing is carried out on this.
   System testing consists of three different kinds of testing activities as described below :

1.

- **Alpha testing:** Alpha testing is the system testing performed by the development team.
- **Beta testing:** Beta testing is the system testing performed by a friendly set of customers.
- **Acceptance testing:** After the software has been delivered, the customer performed acceptance testing to determine whether to accept the delivered software or reject it.

2. **Maintenance:** Maintenance is the most important phase of a software life cycle. The effort spent on maintenance is 60% of the total effort spent to develop a full software. There are basically three types of maintenance :

- **Corrective Maintenance:** This type of maintenance is carried out to correct errors that were not discovered during the product development phase.
- **Perfective Maintenance:** This type of maintenance is carried out to enhance the functionalities of the system based on the customer's request.
- **Adaptive Maintenance:** Adaptive maintenance is usually required for porting the software to work in a new environment such as working on a new computer platform or with a new operating system.

Advantages of Classical Waterfall Model

The classical waterfall model is an idealistic model for software development. It is very simple, so it can be considered the basis for other software development life cycle models. Below are some of the major advantages of this SDLC model:

- This model is very simple and is easy to understand.
- Phases in this model are processed one at a time.
- Each stage in the model is clearly defined.
- This model has very clear and well-understood milestones.
- Process, actions and results are very well documented.
- Reinforces good habits: define-before- design, design-before-code.
- This model works well for smaller projects and projects where requirements are well understood.

Drawbacks of Classical Waterfall Model

The classical waterfall model suffers from various shortcomings, basically, we can't use it in real projects, but we use other software development lifecycle models which are based on the classical waterfall model.

Project Scheduling

Project-task scheduling is a significant project planning activity. It comprises deciding which functions would be taken up when. To schedule the project plan, a software project manager wants to do the following:

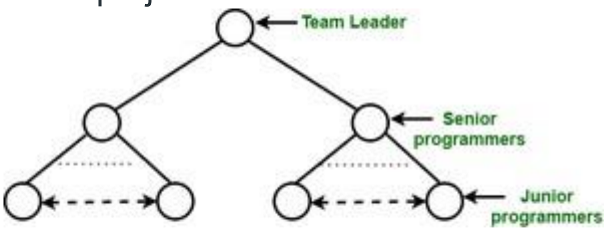
1. Identify all the functions required to complete the project.
2. Break down large functions into small activities.
3. Determine the dependency among various activities.
4. Establish the most likely size for the time duration required to complete the activities.
5. Allocate resources to activities.
6. Plan the beginning and ending dates for different activities.
7. Determine the critical path. A critical way is the group of activities that decide the duration of the project.

Organization and Team structure

There are many ways to organize the project team. Some important ways are as follows :

1. Hierarchical team organization
2. Chief-programmer team organization
3. Matrix team, organization
4. Egoless team organization
5. Democratic team organization 6. **Hierarchical team organization :**

   In this, the people of organization at different levels following a tree structure. People at bottom level generally possess most detailed knowledge about the system. People at higher levels have broader appreciation of the whole project.



Benefits of hierarchical team organization :

- It limits the number of communication paths and stills allows for the needed communication.
- It can be expanded over multiple levels.
- It is well suited for the development of the hierarchical software products.
- Large software projects may have several levels.

Limitations of hierarchical team organization :

- As information has to be travel up the levels, it may get distorted.
- Levels in the hierarchy often judges people socially and financially. Most technical competent programmers tend to be promoted to the management positions which may result in loss of good programmer and also bad manager.

Chief-programmer team organization :

This team organization is composed of a small team consisting the following team members :

- **The Chief programmer :** It is the person who is actively involved in the planning, specification and design process and ideally in the implementation process as well.
- **The project assistant :** It is the closest technical co-worker of the chief programmer.
- **The project secretary :** It relieves the chief programmer and all other programmers of administration tools.
- **Specialists :** These people select the implementation language, implement individual system components and employ software tools and carry out tasks.

Advantages of Chief-programmer team organization :

- Centralized decision-making
- Reduced communication paths
- Small teams are more productive than large teams
- The chief programmer is directly involved in system development and can exercise the better control function.

Disadvantages of Chief-programmer team organization :

- Project survival depends on one person only.
- Can cause the psychological problems as the “chief programmer” is like the “king” who takes all the credit and other members are resentful.
- Team organization is limited to only small team and small team cannot handle every project.

- Effectiveness of team is very sensitive to Chief programmer’s technical and managerial activities.

Matrix Team Organization :

In matrix team organization, people are divided into specialist groups. Each group has a manager. Example of Metric team organization is as follows :

Egoless Team Organization :

Egoless programming is a state of mind in which programmer are supposed to separate themselves from their product. In this team organization goals are set and decisions are made by group consensus. Here group, ‘leadership’ rotates based on tasks to be performed and differing abilities of members.

In this organization work products are discussed openly and all freely examined all team members. There is a major risk which such organization, if teams are composed of inexperienced or incompetent members.

Democratic Team Organization :
It is quite similar to the egoless team organization, but one member is the team leader with some responsibilities :

- Coordination
- Final decisions, when consensus cannot be reached.

Advantages of Democratic Team Organization :

- Each member can contribute to decisions.
- Members can learn from each other.
- Improved job satisfaction.

Disadvantages of Democratic Team Organization :

- Communication overhead increased.
- Need for compatibility of members.
- Less individual responsibility and authority.

Staffing:
Staffing is that part of management which is concerned with obtaining, utilizing, and maintaining capable people to fill all positions in the organization from toplevel to bottom level. It involves the scientific and systematic procurement, allocation, utilization, conservation, and development of human resources. It is the art of acquiring, developing, and maintaining a satisfactory and satisfied workforce. Staffing is that function by which a manager builds an organization through the recruitment, selection, and development of the individual, which also includes a series of activities. It ensures that the organization has the right number of people at the right places, at the right time, and performing the right thing.

Components of Staffing-
There are three aspects or components of staffing, namely, recruitment, selection, and training. They are defined below:

Recruitment: It is the process of finding potential candidates for a particular job in an organization. The process of recruitment involves persuading people to apply for the available positions in the organization.
Selection: It is the process of recognizing potential and hiring the best people out of several possible candidates. This is done by shortlisting and choosing the deserving and eliminating those who are not suitable for the job.
Training: It is the process that involves providing the employees with an idea of the type of work they are supposed to do and how it is to be done. It is a way of keeping the employees updated on the way of work in an organization and the new and advanced technologies.

Risk Management:
A computer code project may be laid low with an outsized sort of risk. so as to be ready to consistently establish the necessary risks which could have an effect on a computer code project, it's necessary to reason risks into completely different categories. The project manager will then examine the risks from every category square measure relevant to the project.
There square measure 3 main classes of risks that may have an effect on a computer code project:

1**Project Risks:**
Project risks concern various sorts of monetary funds, schedules, personnel, resource, and customer-related issues. a vital project risk is schedule slippage. Since computer code is intangible, it's terribly tough to observe and manage a computer code project. it's terribly tough to manage one thing that can not be seen. For any producing project, like producing cars, the project manager will see the merchandise taking form.

1. **Technical Risks:**
   Technical risks concern potential style, implementation, interfacing, testing, and maintenance issues. Technical risks conjointly embody ambiguous specifications, incomplete specification, dynamic specification, technical uncertainty, and technical degeneration. Most technical risks occur thanks to the event team's lean information concerning the project.

2. **Business Risks:**
   This type of risk embodies the risks of building a superb product that nobody needs, losing monetary funds or personal commitments, etc.

Configuration Management

A configuration of the product refers not only to the product's constituent but also to a particular version of the component.

Therefore, SCM is the discipline which

○ Identify change ○ Monitor and control change ○ Ensure the proper implementation of change made to the item. ○ Auditing and reporting on the change made.

Configuration Management (CM) is a technic of identifying, organizing, and controlling modification to software being built by a programming team.

Importance of SCM

It is practical in controlling and managing the access to various SCIs e.g., by preventing the two members of a team for checking out the same component for modification at the same time.

It provides the tool to ensure that changes are being properly implemented.

It has the capability of describing and storing the various constituent of software.

SCM is used in keeping a system in a consistent state by automatically producing derived version upon modification of the same component.

Software project management

Software Project Management

The job pattern of an IT company engaged in software development can be seen split in two parts:

- Software Creation
- Software Project Management

A project is well-defined task, which is a collection of several operations done in order to achieve a goal (for example, software development and delivery). A Project can be characterized as:

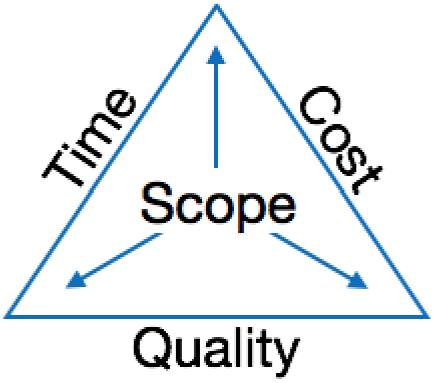
- Every project may has a unique and distinct goal.
- Project is not routine activity or day-to-day operations.
- Project comes with a start time and end time.
- Project ends when its goal is achieved hence it is a temporary phase in the lifetime of an organization.
- Project needs adequate resources in terms of time, manpower, finance, material and knowledge-bank.

Software Project

A Software Project is the complete procedure of software development from requirement gathering to testing and maintenance, carried out according to the execution methodologies, in a specified period of time to achieve intended software product.

Need of software project management

Software is said to be an intangible product. Software development is a kind of all new stream in world business and there's very little experience in building software products. Most software products are tailor made to fit client's requirements. The most important is that the underlying technology changes and advances so frequently and rapidly that experience of one product may not be applied to the other one. All such business and environmental constraints bring risk in software development hence it is essential to manage software projects efficiently.



The image above shows triple constraints for software projects. It is an essential part of software organization to deliver quality product, keeping the cost within client's budget constrain and deliver the project as per scheduled. There are several factors, both internal and external, which may impact this triple constrain triangle. Any of three factor can severely impact the other two.

Therefore, software project management is essential to incorporate user requirements along with budget and time constraints.

Software Project Manager

A software project manager is a person who undertakes the responsibility of executing the software project. Software project manager is thoroughly aware of all the phases of SDLC that the software would go through. Project manager may never directly involve in producing the end product but he controls and manages the activities involved in production.

A project manager closely monitors the development process, prepares and executes various plans, arranges necessary and adequate resources, maintains communication among all team members in order to address issues of cost, budget, resources, time, quality and customer satisfaction.

Let us see few responsibilities that a project manager shoulders -

Managing People

- Act as project leader

- Liaison with stakeholders
- Managing human resources Setting up reporting hierarchy etc.

Managing Project

- Defining and setting up project scope
- Managing project management activities
- Monitoring progress and performance
- Risk analysis at every phase
- Take necessary step to avoid or come out of problems
- Act as project spokesperson

Software Management Activities

Software project management comprises of a number of activities, which contains planning of project, deciding scope of software product, estimation of cost in various terms, scheduling of tasks and events, and resource management. Project management activities may include:

- **Project Planning**
- **Scope Management**
- **Project Estimation**

Project Planning

Software project planning is task, which is performed before the production of software actually starts. It is there for the software production but involves no concrete activity that has any direction connection with software production; rather it is a set of multiple processes, which facilitates software production. Project planning may include the following:

Scope Management

It defines the scope of project; this includes all the activities, process need to be done in order to make a deliverable software product. Scope management is essential because it creates boundaries of the project by clearly defining what would be done in the project and what would not be done. This makes project to contain limited and quantifiable tasks, which can easily be documented and in turn avoids cost and time overrun.

During Project Scope management, it is necessary to -

- Define the scope
- Decide its verification and control
- Divide the project into various smaller parts for ease of management.
- Verify the scope
- Control the scope by incorporating changes to the scope

Project Estimation

For an effective management accurate estimation of various measures is a must. With correct estimation managers can manage and control the project more efficiently and effectively.

Project estimation may involve the following:

- **Software size estimation**
  Software size may be estimated either in terms of KLOC (Kilo Line of Code) or by calculating number of function points in the software. Lines of code depend upon coding practices and Function points vary according to the user or software requirement.
- **Effort estimation**
  The managers estimate efforts in terms of personnel requirement and man-hour required to produce the software. For effort estimation software size should be known. This can either be derived by managers' experience, organization's historical data or software size can be converted into efforts by using some standard formulae.
- **Time estimation**
  Once size and efforts are estimated, the time required to produce the software can be estimated. Efforts required is segregated into sub categories as per the requirement specifications and interdependency of various components of software. Software tasks are divided into smaller tasks, activities or events by Work Breakthrough Structure (WBS). The tasks are scheduled on day-to-day basis or in calendar months.
  The sum of time required to complete all tasks in hours or days is the total time invested to complete the project.
- **Cost estimation**
  This might be considered as the most difficult of all because it depends on more elements than any of the previous ones. For estimating project cost, it is required to consider - ○ Size of software ○ Software quality ○ Hardware ○ Additional software or tools, licenses etc. ○ Skilled personnel with task-specific skills ○ Travel involved ○ Communication
  ○ Training and support

Project Estimation Techniques

We discussed various parameters involving project estimation such as size, effort, time and cost.

Project manager can estimate the listed factors using two broadly recognized techniques –

Decomposition Technique

This technique assumes the software as a product of various compositions.

There are two main models -

- **Line of Code** Estimation is done on behalf of number of line of codes in the software product.
- **Function Points** Estimation is done on behalf of number of function points in the software product.

Empirical Estimation Technique

This technique uses empirically derived formulae to make estimation.These formulae are based on LOC or FPs.

- **Putnam Model**
  This model is made by Lawrence H. Putnam, which is based on Norden's frequency distribution (Rayleigh curve). Putnam model maps time and efforts required with software size.
- **COCOMO**
  COCOMO stands for COnstructive COst MOdel, developed by Barry W. Boehm. It divides the software product into three categories of software: organic, semidetached and embedded.

Project Scheduling

Project Scheduling in a project refers to roadmap of all activities to be done with specified order and within time slot allotted to each activity. Project managers tend to define various tasks, and project milestones and arrange them keeping various factors in mind. They look for tasks lie in critical path in the schedule, which are necessary to complete in specific manner (because of task interdependency) and strictly within the time allocated. Arrangement of tasks which lies out of critical path are less likely to impact over all schedule of the project.

For scheduling a project, it is necessary to -

- Break down the project tasks into smaller, manageable form
- Find out various tasks and correlate them
- Estimate time frame required for each task
- Divide time into work-units
- Assign adequate number of work-units for each task
- Calculate total time required for the project from start to finish

Resource management

All elements used to develop a software product may be assumed as resource for that project. This may include human resource, productive tools and software libraries.

The resources are available in limited quantity and stay in the organization as a pool of assets. The shortage of resources hampers the development of project and it can lag behind the schedule. Allocating extra resources increases development cost in the end. It is therefore necessary to estimate and allocate adequate resources for the project.

Resource management includes -

- Defining proper organization project by creating a project team and allocating responsibilities to each team member
- Determining resources required at a particular stage and their availability
- Manage Resources by generating resource request when they are required and deallocating them when they are no more needed.

Project Risk Management

Risk management involves all activities pertaining to identification, analyzing and making provision for predictable and non-predictable risks in the project. Risk may include the following:

- Experienced staff leaving the project and new staff coming in.
- Change in organizational management.
- Requirement change or misinterpreting requirement.
- Under-estimation of required time and resources. Technological changes, environmental changes, business competition.

Risk Management Process

There are following activities involved in risk management process:

- **Identification -** Make note of all possible risks, which may occur in the project.
- **Categorize -** Categorize known risks into high, medium and low risk intensity as per their possible impact on the project.
- **Manage -** Analyze the probability of occurrence of risks at various phases. Make plan to avoid or face risks. Attempt to minimize their side-effects.
- **Monitor -** Closely monitor the potential risks and their early symptoms. Also monitor the effects of steps taken to mitigate or avoid them.

Project Execution & Monitoring

In this phase, the tasks described in project plans are executed according to their schedules.

Execution needs monitoring in order to check whether everything is going according to the plan. Monitoring is observing to check the probability of risk and taking measures to address the risk or report the status of various tasks.

These measures include -

- **Activity Monitoring -** All activities scheduled within some task can be monitored on dayto-day basis. When all activities in a task are completed, it is considered as complete.
- **Status Reports -** The reports contain status of activities and tasks completed within a given time frame, generally a week. Status can be marked as finished, pending or work-inprogress etc.
- **Milestones Checklist -** Every project is divided into multiple phases where major tasks are performed (milestones) based on the phases of SDLC. This milestone checklist is prepared once every few weeks and reports the status of milestones.

Project Communication Management

Effective communication plays vital role in the success of a project. It bridges gaps between client and the organization, among the team members as well as other stake holders in the project such as hardware suppliers.

Communication can be oral or written. Communication management process may have the following steps:

- **Planning** - This step includes the identifications of all the stakeholders in the project and the mode of communication among them. It also considers if any additional communication facilities are required.

- **Sharing -** After determining various aspects of planning, manager focuses on sharing correct information with the correct person on correct time. This keeps every one involved the project up to date with project progress and its status.
- **Feedback** - Project managers use various measures and feedback mechanism and create status and performance reports. This mechanism ensures that input from various stakeholders is coming to the project manager as their feedback.
- **Closure** - At the end of each major event, end of a phase of SDLC or end of the project itself, administrative closure is formally announced to update every stakeholder by sending email, by distributing a hardcopy of document or by other mean of effective communication.

After closure, the team moves to next phase or project.

Configuration Management

Configuration management is a process of tracking and controlling the changes in software in terms of the requirements, design, functions and development of the product.

IEEE defines it as "the process of identifying and defining the items in the system, controlling the change of these items throughout their life cycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items".

Generally, once the SRS is finalized there is less chance of requirement of changes from user. If they occur, the changes are addressed only with prior approval of higher management, as there is a possibility of cost and time overrun.

Baseline

A phase of SDLC is assumed over if it baselined, i.e. baseline is a measurement that defines completeness of a phase. A phase is baselined when all activities pertaining to it are finished and well documented. If it was not the final phase, its output would be used in next immediate phase.

Configuration management is a discipline of organization administration, which takes care of occurrence of any change (process, requirement, technological, strategical etc.) after a phase is baselined. CM keeps check on any changes done in software.

Change Control

Change control is function of configuration management, which ensures that all changes made to software system are consistent and made as per organizational rules and regulations.

A change in the configuration of product goes through following steps -

- **Identification** - A change request arrives from either internal or external source. When change request is identified formally, it is properly documented.
- **Validation** - Validity of the change request is checked and its handling procedure is confirmed.
- **Analysis** - The impact of change request is analyzed in terms of schedule, cost and required efforts. Overall impact of the prospective change on system is analyzed.
- **Control** - If the prospective change either impacts too many entities in the system or it is unavoidable, it is mandatory to take approval of high authorities before change is incorporated into the system. It is decided if the change is worth incorporation or not. If it is not, change request is refused formally.

- **Execution** - If the previous phase determines to execute the change request, this phase take appropriate actions to execute the change, does a thorough revision if necessary.
- **Close request** - The change is verified for correct implementation and merging with the rest of the system. This newly incorporated change in the software is documented properly and the request is formally is closed.

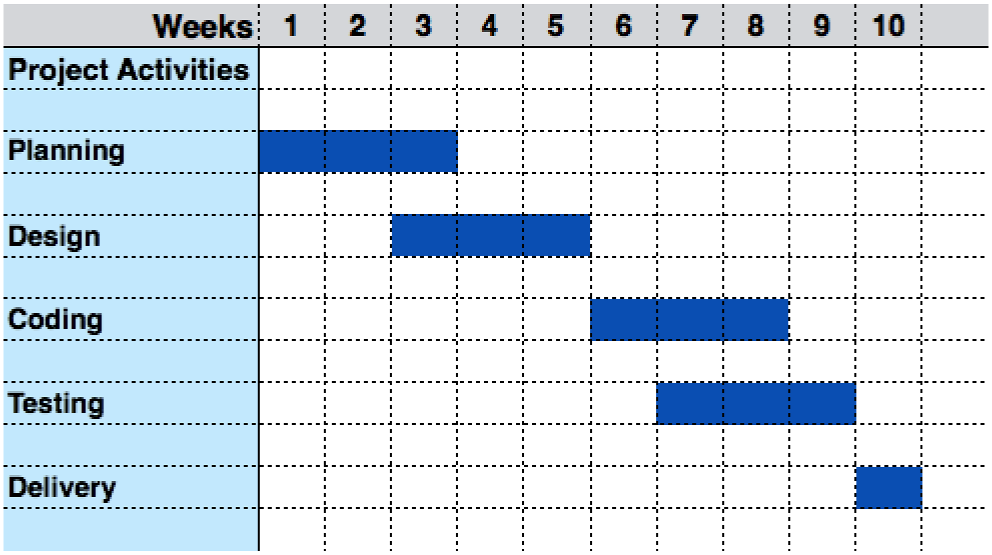
Project Management Tools

The risk and uncertainty rises multifold with respect to the size of the project, even when the project is developed according to set methodologies.

There are tools available, which aid for effective project management. A few are described -

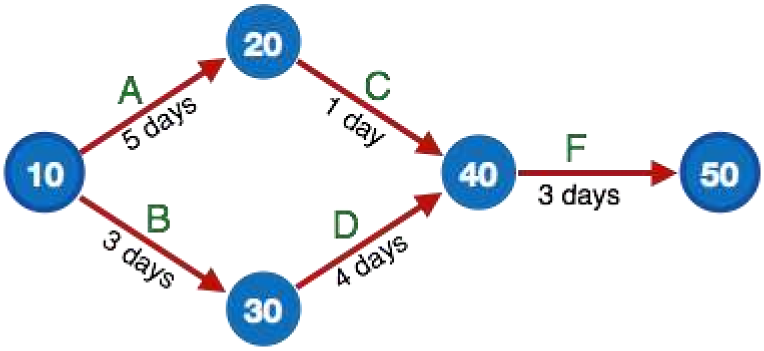
Gantt Chart

Gantt charts was devised by Henry Gantt (1917). It represents project schedule with respect to time periods. It is a horizontal bar chart with bars representing activities and time scheduled for the project activities.



PERT Chart

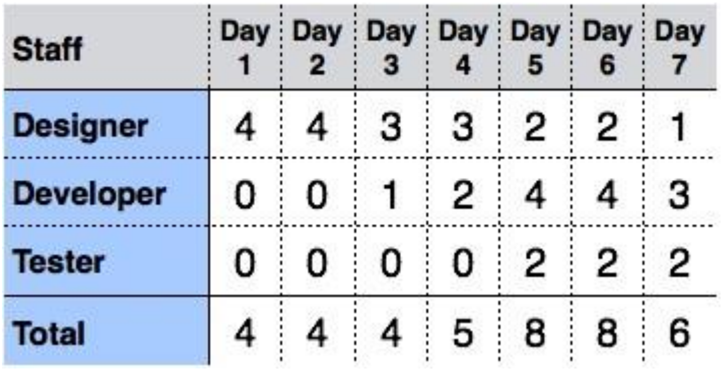
PERT (Program Evaluation & Review Technique) chart is a tool that depicts project as network diagram. It is capable of graphically representing main events of project in both parallel and consecutive way. Events, which occur one after another, show dependency of the later event over the previous one.



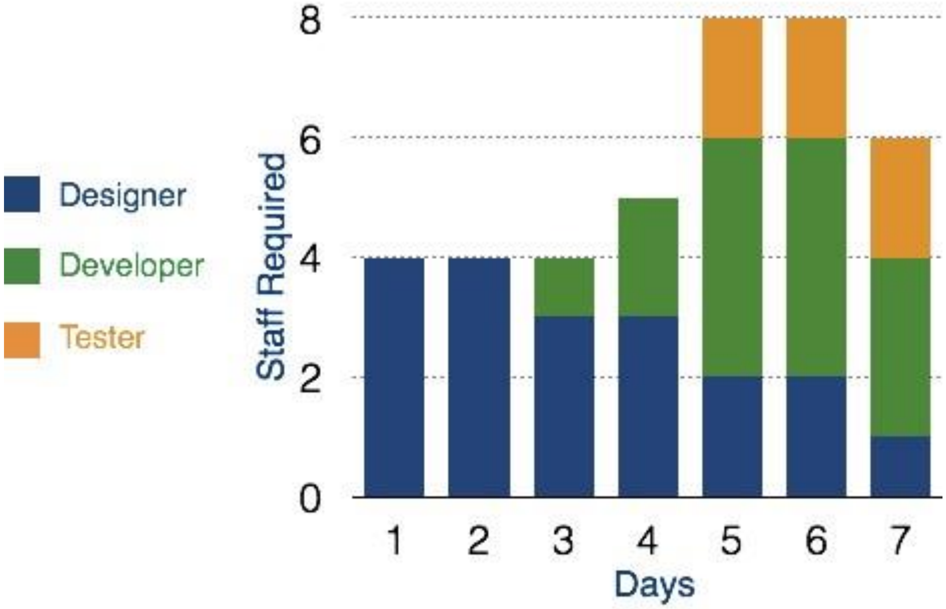
Events are shown as numbered nodes. They are connected by labeled arrows depicting sequence of tasks in the project.

Resource Histogram

This is a graphical tool that contains bar or chart representing number of resources (usually skilled staff) required over time for a project event (or phase). Resource Histogram is an effective tool for staff planning and coordination.



Staff	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
Designer	4	4	3	3	2	2	1
Developer	0	0	1	2	4	4	3
Tester	0	0	0	0	2	2	2
Total	4	4	4	5	8	8	6



Critical Path Analysis

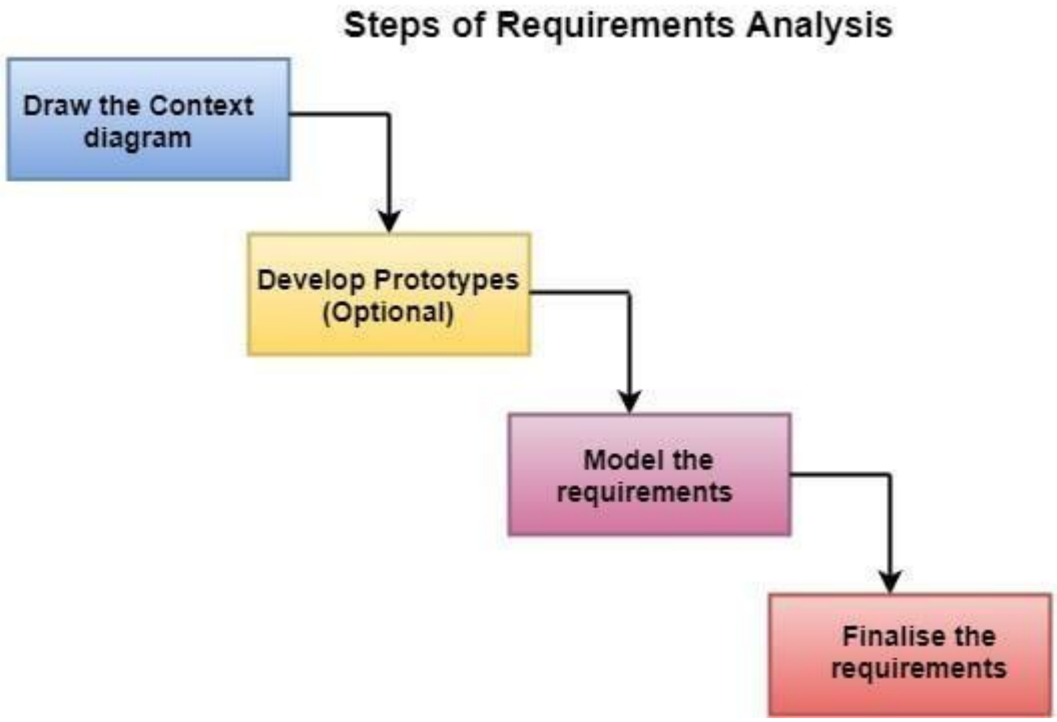
This tools is useful in recognizing interdependent tasks in the project. It also helps to find out the shortest path or critical path to complete the project successfully. Like PERT diagram, each event is allotted a specific time frame. This tool shows dependency of event assuming an event can proceed to next only if the previous one is completed.

The events are arranged according to their earliest possible start time. Path between start and end node is critical path which cannot be further reduced and all events require to be executed in same order.

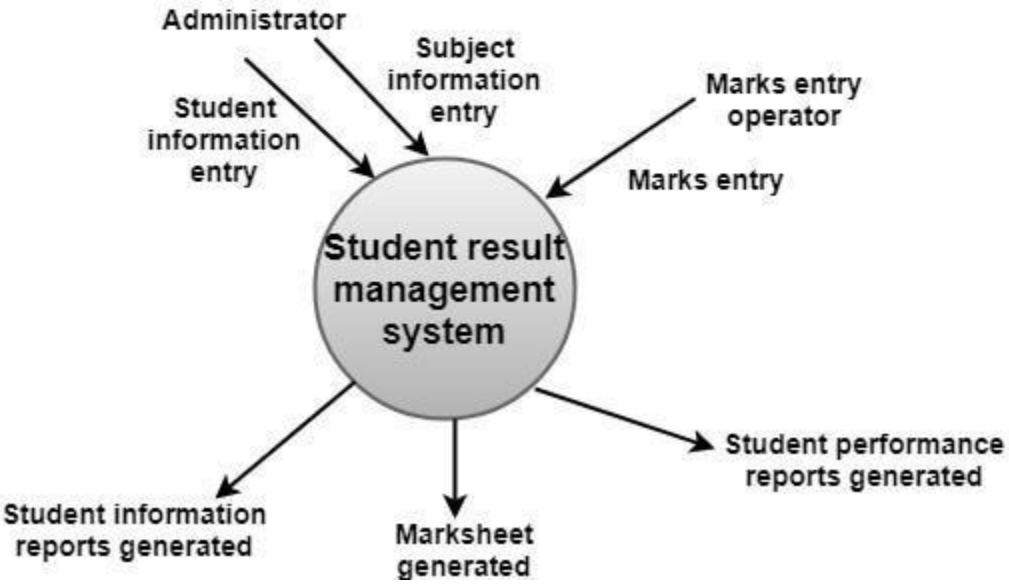
Requirements Analysis and specification

Requirement analysis is significant and essential activity after elicitation. We analyze, refine, and scrutinize the gathered requirements to make consistent and unambiguous requirements. This activity reviews all requirements and may provide a graphical view of the entire system. After the completion of the analysis, it is expected that the understandability of the project may improve significantly. Here, we may also use the interaction with the customer to clarify points of confusion and to understand which requirements are more important than others.

The various steps of requirement analysis are shown in fig:


Steps of Requirements Analysis
Draw the Context diagram
Develop Prototypes (Optional)
Model the requirements
Finalise the requirements

(i) Draw the context diagram: The context diagram is a simple model that defines the boundaries and interfaces of the proposed systems with the external world. It identifies the entities outside the proposed system that interact with the system. The context diagram of student result management system is given below:


Administrator
Student information entry
Subject information entry
Marks entry operator
Marks entry
Student result management system
Student information reports generated
Marksheet generated
Student performance reports generated

(ii) Development of a Prototype (optional): One effective way to find out what the customer wants is to construct a prototype, something that looks and preferably acts as part of the system they say they want.

We can use their feedback to modify the prototype until the customer is satisfied continuously. Hence, the prototype helps the client to visualize the proposed system and increase the understanding of the requirements. When developers and users are not sure about some of the elements, a prototype may help both the parties to take a final decision.

Some projects are developed for the general market. In such cases, the prototype should be shown to some representative sample of the population of potential purchasers. Even though a person who tries out a prototype may not buy the final system, but their feedback may allow us to make the product more attractive to others.

The prototype should be built quickly and at a relatively low cost. Hence it will always have limitations and would not be acceptable in the final system. This is an optional activity.

(iii) Model the requirements: This process usually consists of various graphical representations of the functions, data entities, external entities, and the relationships between them. The graphical view may help to find incorrect, inconsistent, missing, and superfluous requirements. Such models include the Data Flow diagram, Entity-Relationship diagram, Data Dictionaries, Statetransition diagrams, etc.

(iv) Finalise the requirements: After modeling the requirements, we will have a better understanding of the system behavior. The inconsistencies and ambiguities have been identified and corrected. The flow of data amongst various modules has been analyzed. Elicitation and analyze activities have provided better insight into the system. Now we finalize the analyzed requirements, and the next step is to document these requirements in a prescribed format.

Software Requirement Specification (SRS)

In order to form a good SRS, here you will see some points which can be used and should be considered to form a structure of good SRS. These are as follows :

1. Introduction

- **(i)** Purpose of this document
- **(ii)** Scope of this document
- **(iii)** Overview

2. General description
3. Functional Requirements
4. Interface Requirements
5. Performance Requirements
6. Design Constraints
7. Non-Functional Attributes
8. Preliminary Schedule and Budget
9. Appendices

Software Requirement Specification (SRS) Format as name suggests, is complete specification and description of requirements of software that needs to be fulfilled for successful development of software system. These requirements can be functional as well as non-functional depending upon type of requirement. The interaction between different customers and contractor is done because its necessary to fully understand needs of customers. Depending upon information gathered after interaction, SRS is developed which describes requirements of software that may include changes and modifications that is needed to be done to increase quality of product and to satisfy customer's demand.

1. **Introduction :**
   - **(i) Purpose of this Document –**
     At first, main aim of why this document is necessary and what's purpose of document is explained and described.
   - **(ii) Scope of this document –**
     In this, overall working and main objective of document and what value it will provide to customer is described and explained. It also includes a description of development cost and time required.
   - **(iii) Overview –**
     In this, description of product is explained. It's simply summary or overall review of product.

2. **General description :**
   In this, general functions of product which includes objective of user, a user characteristic, features, benefits, about why its importance is mentioned. It also describes features of user community.

3. **Functional Requirements :**

In this, possible outcome of software system which includes effects due to operation of program is fully explained. All functional requirements which may include calculations, data processing, etc. are placed in a ranked order.

4. **Interface Requirements :**
   In this, software interfaces which mean how software program communicates with each other or users either in form of any language, code, or message are fully described and explained. Examples can be shared memory, data streams, etc.

5. **Performance Requirements :**
   In this, how a software system performs desired functions under specific condition is explained. It also explains required time, required memory, maximum error rate, etc.

6. **Design Constraints :**
   In this, constraints which simply means limitation or restriction are specified and explained for design team. Examples may include use of a particular algorithm, hardware and software limitations, etc.

7. **Non-Functional Attributes :**
   In this, non-functional attributes are explained that are required by software system for better performance. An example may include Security, Portability, Reliability, Reusability, Application compatibility, Data integrity, Scalability capacity, etc.

8. **Preliminary Schedule and Budget :**
   In this, initial version and budget of project plan are explained which include overall time duration required and overall cost required for development of project

9. **Appendices :**
   In this, additional information like references from where information is gathered, definitions of some specific terms, acronyms, abbreviations, etc. are given and explained.

Software Design

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages.

Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fulfill the requirements mentioned in SRS.

Software Design Levels

Software design yields three levels of results:

- **Architectural Design -** The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.
- **High-level Design-** The high-level design breaks the ‘single entity-multiple component’ concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on

how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.

- **Detailed Design-** Detailed design deals with the implementation part of what is seen as a system and its subsystems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

Modularization

Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.

Modular design unintentionally follows the rules of ‘divide and conquer’ problem-solving strategy this is because there are many other benefits attached with the modular design of a software. Advantage of modularization:

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be re-used again
- Concurrent execution can be made possible
- Desired from security aspect

Concurrency

Back in time, all software are meant to be executed sequentially. By sequential execution we mean that the coded instruction will be executed one after another implying only one portion of program being activated at any given time. Say, a software has multiple modules, then only one of all the modules can be found active at any time of execution.

In software design, concurrency is implemented by splitting the software into multiple independent units of execution, like modules and executing them in parallel. In other words, concurrency provides capability to the software to execute more than one part of code in parallel to each other.

It is necessary for the programmers and designers to recognize those modules, which can be made parallel execution.

Example

The spell check feature in word processor is a module of software, which runs along side the word processor itself.

Coupling and Cohesion

When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are set of instructions put together in order to achieve some tasks. They are though, considered as single entity but may refer to each other to work together. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

Cohesion

Cohesion is a measure that defines the degree of intradependability within elements of a module. The greater the cohesion, the better is the program design.

There are seven types of cohesion, namely –

- **Co-incidental cohesion -** It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.
- **Logical cohesion -** When logically categorized elements are put together into a module, it is called logical cohesion.
- **Temporal Cohesion -** When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.
- **Procedural cohesion -** When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.
- **Communicational cohesion -** When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.
- **Sequential cohesion -** When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.
- **Functional cohesion -** It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

Coupling

Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program.

There are five levels of coupling, namely -

- **Content coupling -** When a module can directly access or modify or refer to the content of another module, it is called content level coupling.
- **Common coupling-** When multiple modules have read and write access to some global data, it is called common or global coupling.
- **Control coupling-** Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.
- **Stamp coupling-** When multiple modules share common data structure and work on different part of it, it is called stamp coupling.
- **Data coupling-** Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Ideally, no coupling is considered to be the best.

Design Verification

The output of software design process is design documentation, pseudo codes, detailed logic diagrams, process diagrams, and detailed description of all functional or non-functional requirements.

The next phase, which is the implementation of software, depends on all outputs mentioned above.

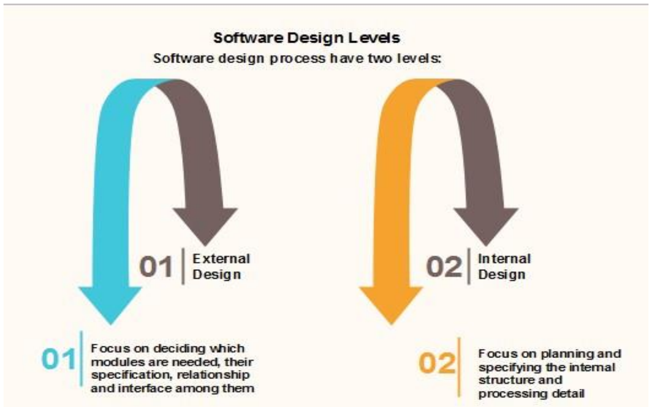
It is then becomes necessary to verify the output before proceeding to the next phase. The early any mistake is detected, the better it is or it might not be detected until testing of the product. If the outputs of design phase are in formal notation form, then their associated tools for verification should be used otherwise a thorough design review can be used for verification and validation.

By structured verification approach, reviewers can detect defects that might be caused by overlooking some conditions. A good design review is important for good software design, accuracy and quality.

Software Design

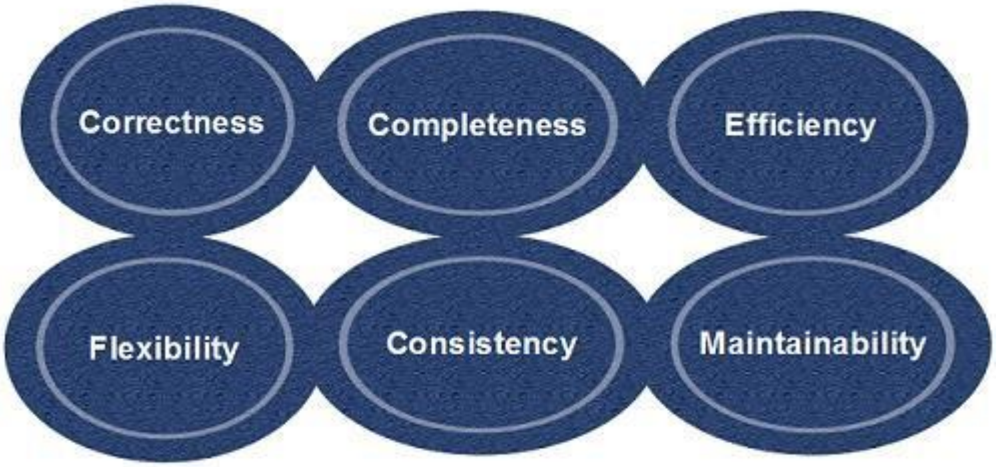
Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily implementable using programming language.

The software design phase is the first step in **SDLC (Software Design Life Cycle)**, which moves the concentration from the problem domain to the solution domain. In software design, we consider the system to be a set of components or modules with clearly defined behaviors & boundaries.


Software Design Levels
Software design process have two levels:
01 External Design
02 Internal Design
01 Focus on deciding which modules are needed, their specification, relationship and interface among them
02 Focus on planning and specifying the internal structure and processing detail

Objectives of Software Design

Following are the purposes of Software design:



Objectives of Software Design

1. **Correctness:**Software design should be correct as per requirement.
2. **Completeness:**The design should have all components like data structures, modules, and external interfaces, etc.
3. **Efficiency:**Resources should be used efficiently by the program.
4. **Flexibility:**Able to modify on changing needs.
5. **Consistency:**There should not be any inconsistency in the design.
6. **Maintainability:** The design should be so simple so that it can be easily maintainable by other designers.

Cohesion & coupling

These two topics - coupling and cohesion, have to do with the quality of an OO design.
In general, good OO design calls for loose coupling and shuns tight coupling.
Good OO design calls for high cohesion, and shuns low cohesion.
What is coupling?

Coupling is the degree to which one class knows about another class.

What is loose coupling?

If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled that is a a good thing.

What is tight coupling?

If class A relies on parts of class B that are not part of class B's interface, then the coupling between the classes is tighten that is not a good thing.
In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.

Coupling example

Imagine what happens when class B is enhanced.
It's quite possible that the developer enhancing class B has no knowledge of class A, why would she?
Class B's developer ought to feel that any enhancements that don't break the class's interface should be safe, so she might change some noninterface part of the class, which then causes class A to break.
At the far end of the coupling spectrum is the horrible situation in which class A knows non-API stuff about class B, and class B knows non-API stuff about class A - this is REALLY BAD. If either class is ever changed, there's a chance that the other class will break.
Let's look at an obvious example of tight coupling, which has been enabled by poor encapsulation:

```
// Tightly coupled class design - Bad thing

class DoTaxes {

float rate;

float doColorado() {
SalesTaxRates str = new SalesTaxRates();
rate = str.salesRate; // ouch //
this should be a method call:
// rate = str.getSalesRate("CO");
// do stuff with rate
}
}

class SalesTaxRates {

public float salesRate; // should be private public
float adjustedSalesRate; // should be private

public float getSalesRate(String region) {
salesRate = new DoTaxes().doColorado(); // ouch again!
// do region-based calculations return
adjustedSalesRate;
}
}
```

Ideally, all interactions between objects in an OO system should use the APIs.
In other words, the contracts, of the object's respective classes.
Theoretically, if all of the classes in an application have welldesigned APIs, then it should be possible for all interclass interactions to use those APIs exclusively.
What is cohesion?

Cohesion is all about how a single class is designed.
The term cohesion is used to indicate the degree to which a class has a single, well-focused purpose.
Keep in mind that cohesion is a subjective concept.
The more focused a class is, the higher its cohesiveness - a good thing.
Benefit or Advantages of Cohesion

The key benefit of high cohesion is that such classes are typically much easier to maintain (and less frequently changed) than classes with low cohesion.
Another benefit of high cohesion is that classes with a wellfocused purpose tend to be more reusable than other classes.
Cohesion example

Less cohesive class design

```
// Less cohesive class design
class BudgetReport {

void connectToRDBMS() {
}

void generateBudgetReport() {
}

void saveToFile() {
}

void print() {
}
}
```

More cohesive class design

```
// More cohesive class design class
BudgetReport {
Options getReportingOptions() {
}

void generateBudgetReport(Options o) {
}
}

class ConnectToRDBMS {
DBconnection getRDBMS() {
}
}

class PrintStuff {
PrintOptions getPrintOptions() {
}
}

class FileSaver {
SaveOptions getFileSaveOptions() {
}
}
```

This design is much more cohesive because Instead of one class that does everything, we have broken the system into four main classes.
Each with a very specific, or cohesive, role.

Because we have built these specialized, reusable classes.

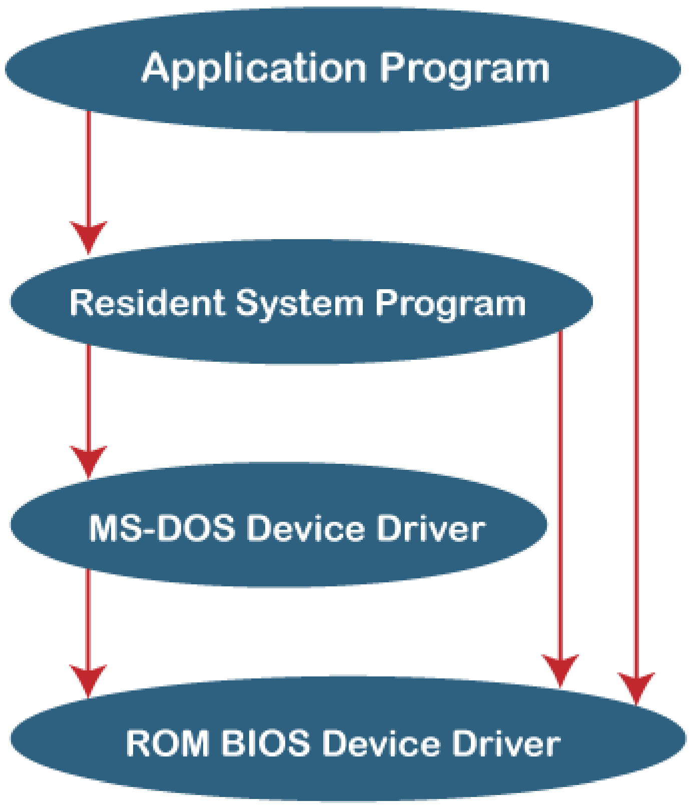
Design approaches in Operating System

The operating system may be implemented with the assistance of several structures. The structure of the operating system is mostly determined by how the many common components of the OS are integrated and merged into the kernel. In this article, you will learn the following structure of the OS. Various structures are used in the design of the operating system. These structures are as follows:

1. **Simple Structure**
2. **Micro-Kernel Structure**
3. **Layered Structure**

Simple Structure

Such OS's are small, simple, and limited, with no well-defined structure. There is a lack of separation between the interfaces and levels of functionality. The MS-DOS is the best example of such an operating system. Application programs in MS-DOS can access basic I/O functions. If one of the user programs fails on these OSs, the complete system crashes. Below is the diagram of the MS-DOS structure that may help you understand the simple structure.



MS-DOS Structure

Advantages and Disadvantages of Simple Structure

There are various advantages and disadvantages of the Simple Structure. Some advantages and disadvantages of the Simple Structure are as follows:

Advantages

1. It provides superior application performance due to the limited interfaces between the application program and the hardware.
2. It is simple for kernel developers to create such an operating system.

Disadvantages

1. The structure is quite complex because there are no apparent boundaries between modules.
2. It does not impose data concealment in the operating system.

Micro-Kernel Structure

This micro-kernel structure creates the OS by eliminating all nonessential kernel components and implementing them as user programs and systems. Therefore, a smaller kernel is known as a micro-kernel.

The benefits of this micro-kernel structure are that all new services must be added to userspace rather than the kernel, and the kernel does not require to be updated. Therefore, it is more secure and trustworthy. If a service fails, the remainder of the OS is unaffected. Mac OS is the best instance of this type of operating system.

Advantages and Disadvantages of Micro-Kernel Structure

There are various advantages and disadvantages of the MicroKernel Structure. Some advantages and disadvantages of the Micro-Kernel Structure are as follows:

Advantages

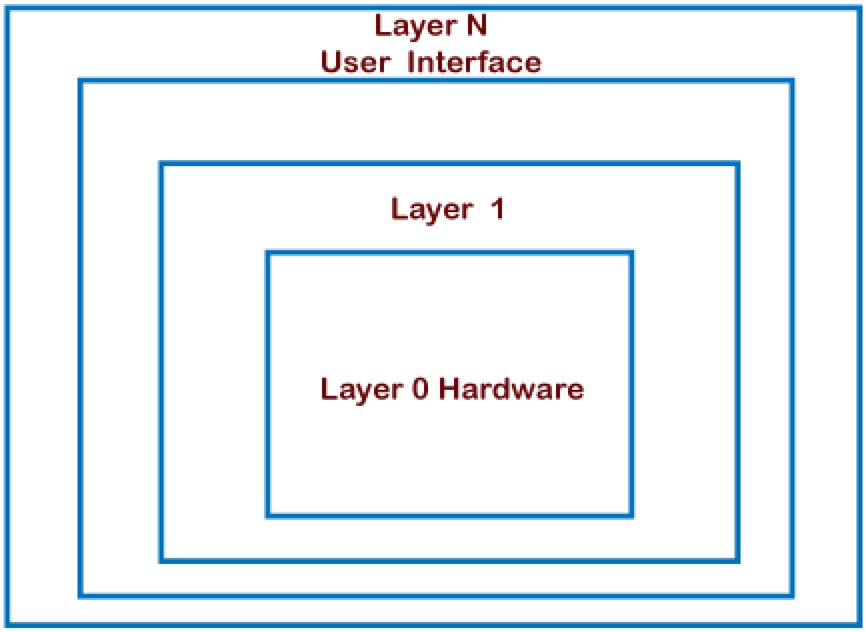
1. It allows the OS to be portable across platforms.
2. They can be effectively tested because the microkernels are small.

Disadvantages

1. The performance of the system suffers as the level of intermodule communication rises.

Layered Structure

An operating system can be divided into sections while retaining far more control over the system. The OS is divided into layers in this arrangement (levels). The hardware is on the ***bottom layer (layer 0)***, and the user interface is on the ***top layer (layer N)***. These layers are designed in such a way that each layer only requires the functions of the lower-level layers. Debugging is simplified because if lower-level layers are debugged, and an error occurs during debugging, the error must occur only on that layer. The lower-level layers have been thoroughly tested.



Layered - Structure

UNIX is the best example of the Layered Structure. The main disadvantage of this structure is that data must be updated and sent on to each layer, which adds overhead to the system. Furthermore, careful planning of the layers is required because a layer may use only lower-level layers.

Advantages and Disadvantages of Layered Structure

There are various advantages and disadvantages of the Layered Structure. Some advantages and disadvantages of the Layered Structure are as follows:

Advantages

1. Layering makes it easier to improve the OS as the implementation of a layer may be changed easily without affecting the other layers.
2. Debugging and system verification are simple to carry out.

Disadvantages

1. When compared to a simple structure, this structure degrades application performance.
2. It needs better planning to construct the layers because higher layers only utilize the functionalities of lower layers.

Modular structure or approach

It is regarded as the best approach for an operating system. It involves designing a modular kernel. It is comparable to a layered structure in that each kernel has specified and protected interfaces, but it is more flexible because a module may call any other module. The kernel contains only a limited number of basic components, and extra services are added to the kernel as dynamically loadable modules either during runtime or at boot time.

Data Flow Diagrams

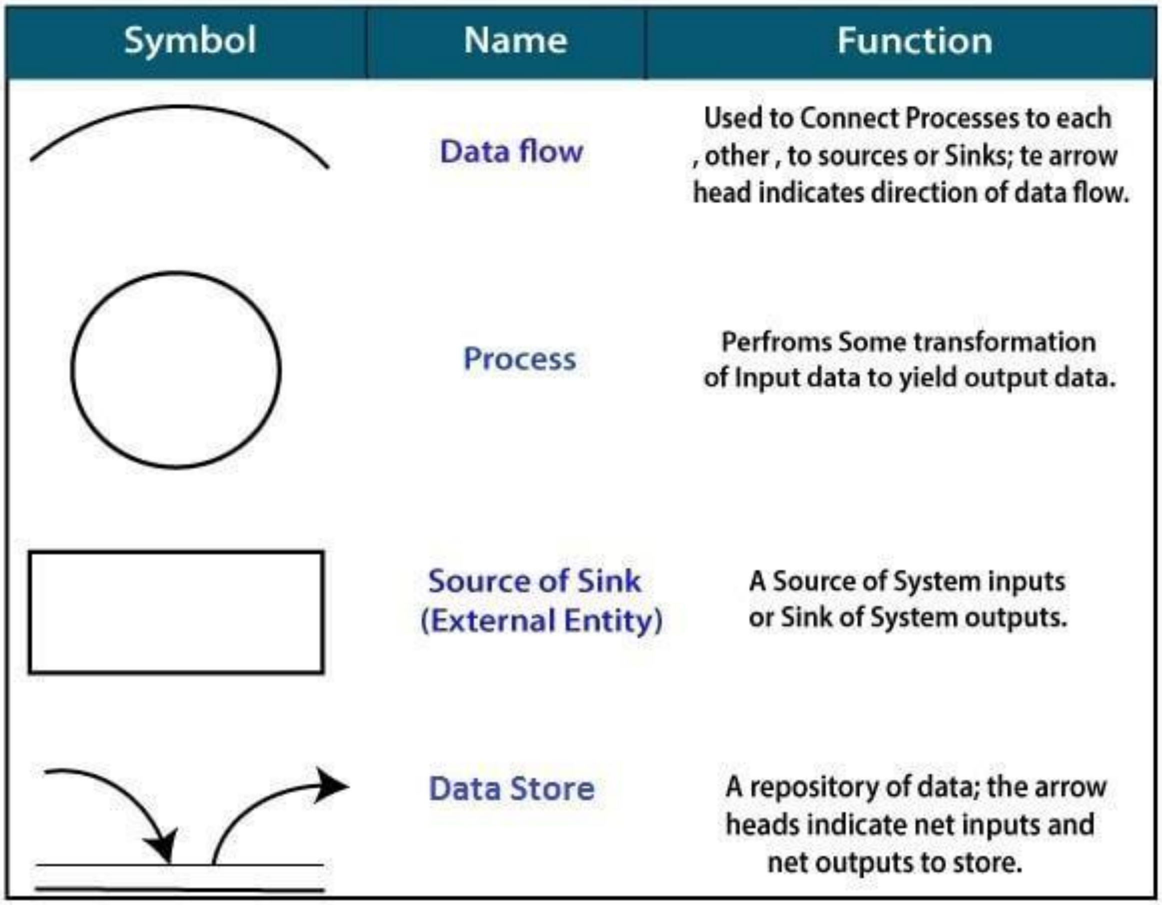
A Data Flow Diagram (DFD) is a traditional visual representation of the information flows within a system. A neat and clear DFD can depict the right amount of the system requirement graphically. It can be manual, automated, or a combination of both.

It shows how data enters and leaves the system, what changes the information, and where data is stored.

The objective of a DFD is to show the scope and boundaries of a system as a whole. It may be used as a communication tool between a system analyst and any person who plays a part in the order that acts as a starting point for redesigning a system. The DFD is also called as a data flow graph or bubble chart.

The following observations about DFDs are essential:

1. All names should be unique. This makes it easier to refer to elements in the DFD.
2. Remember that DFD is not a flow chart. Arrows is a flow chart that represents the order of events; arrows in DFD represents flowing data. A DFD does not involve any order of events.
3. Suppress logical decisions. If we ever have the urge to draw a diamond-shaped box in a DFD, suppress that urge! A diamond-shaped box is used in flow charts to represents decision points with multiple exists paths of which the only one is taken. This implies an ordering of events, which makes no sense in a DFD.
4. Do not become bogged down with details. Defer error conditions and error handling until the end of the analysis. Standard symbols for DFDs are derived from the electric circuit diagram analysis and are shown in fig:



Symbol	Name	Function
	Data flow	Used to Connect Processes to each , other , to sources or Sinks; te arrow head indicates direction of data flow.
	Process	Perfroms Some transformation of Input data to yield output data.
	Source of Sink (External Entity)	A Source of System inputs or Sink of System outputs.
	Data Store	A repository of data; the arrow heads indicate net inputs and net outputs to store.

Symbols for Data Flow Diagrams

Circle: A circle (bubble) shows a process that transforms data inputs into data outputs.

Data Flow: A curved line shows the flow of data into or out of a process or data store.

Data Store: A set of parallel lines shows a place for the collection of data items. A data store indicates that the data is stored which can be used at a later stage or by the other processes in a different order. The data store can have an element or group of elements.

Source or Sink: Source or Sink is an external entity and acts as a source of system inputs or sink of system outputs.

Levels in Data Flow Diagrams (DFD)

The DFD may be used to perform a system or software at any level of abstraction. Infact, DFDs may be partitioned into levels that represent increasing information flow and functional detail. Levels in DFD are numbered 0, 1, 2 or beyond. Here, we will see primarily three levels in the data flow diagram, which are: 0-level DFD, 1-level DFD, and 2-level DFD.

0-level DFDM

It is also known as fundamental system model, or context diagram represents the entire software requirement as a single bubble with input and output data denoted by incoming and outgoing arrows. Then the system is decomposed and described as a DFD with multiple bubbles. Parts of the system represented by each of these bubbles are then decomposed and documented as more and more detailed DFDs. This process may be repeated at as many levels as necessary until the program at hand is well understood. It is essential to preserve the number of inputs and outputs between levels, this concept is called leveling by DeMacro. Thus, if bubble "A" has two inputs x_1 and x_2 and one output y, then the expanded DFD, that represents "A" should have exactly two external inputs and one external output as shown in fig:

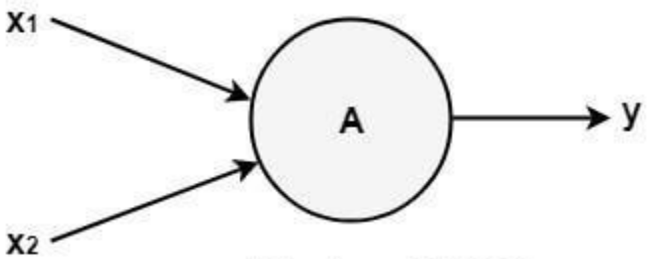


Fig: Level-0 DFD.

The Level-0 DFD, also called context diagram of the result management system is shown in fig. As the bubbles are decomposed into less and less abstract bubbles, the corresponding data flow may also be needed to be decomposed.

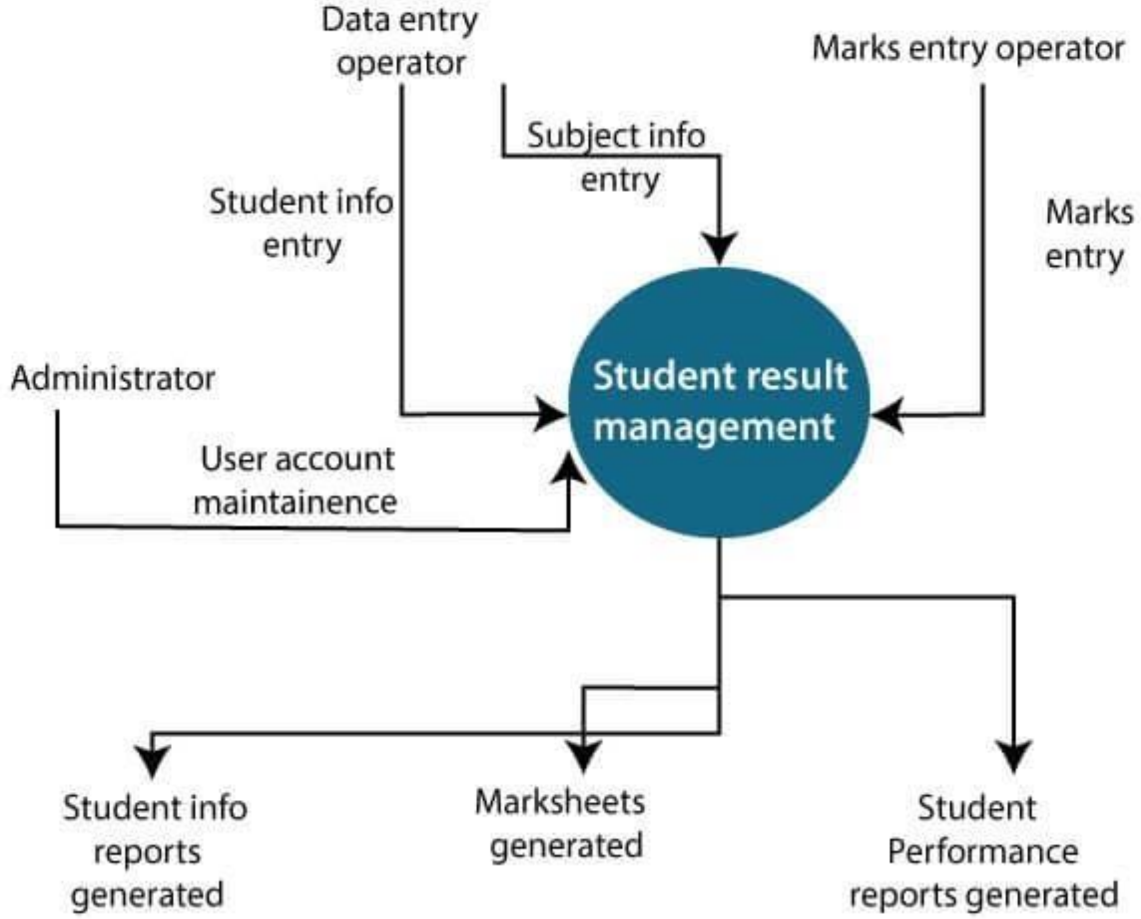


Fig: Level-0 DFD of result management system

1-level DFD

In 1-level DFD, a context diagram is decomposed into multiple bubbles/processes. In this level, we highlight the main objectives of the system and breakdown the high-level process of 0-level DFD into subprocesses.

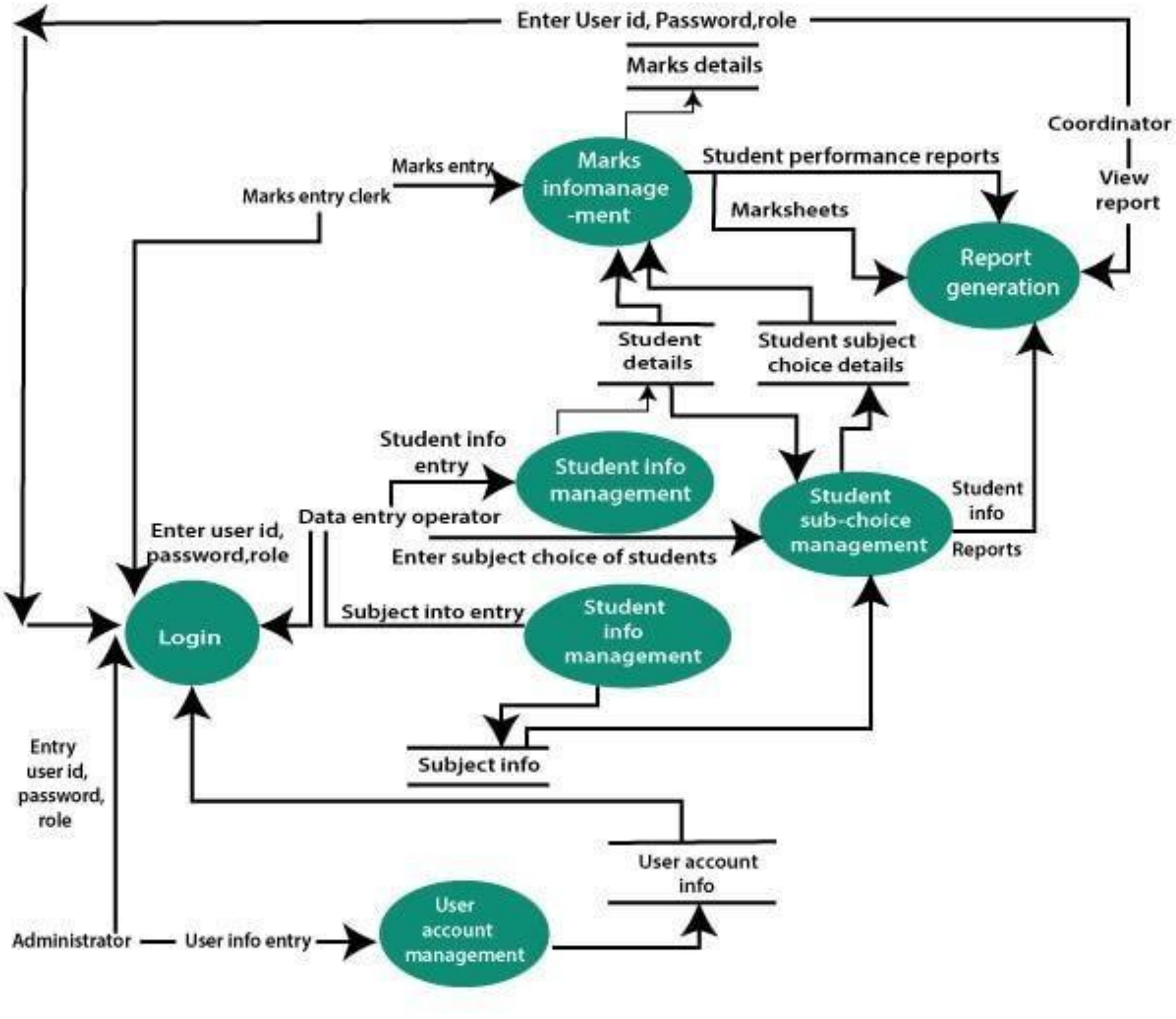
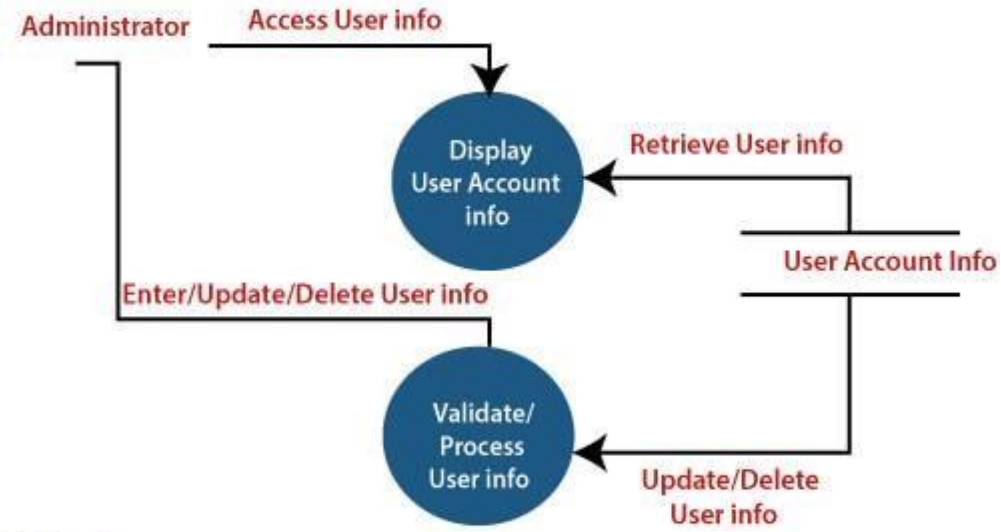


Fig: Level-1 DFD of result management system

2-Level DFD

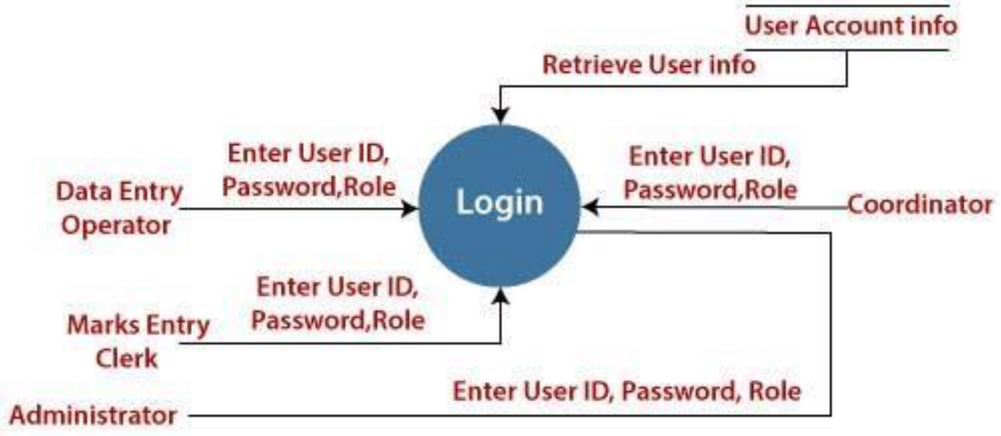
2-level DFD goes one process deeper into parts of 1-level DFD. It can be used to project or record the specific/necessary detail about the system's functioning.

1.User Account Maintenance

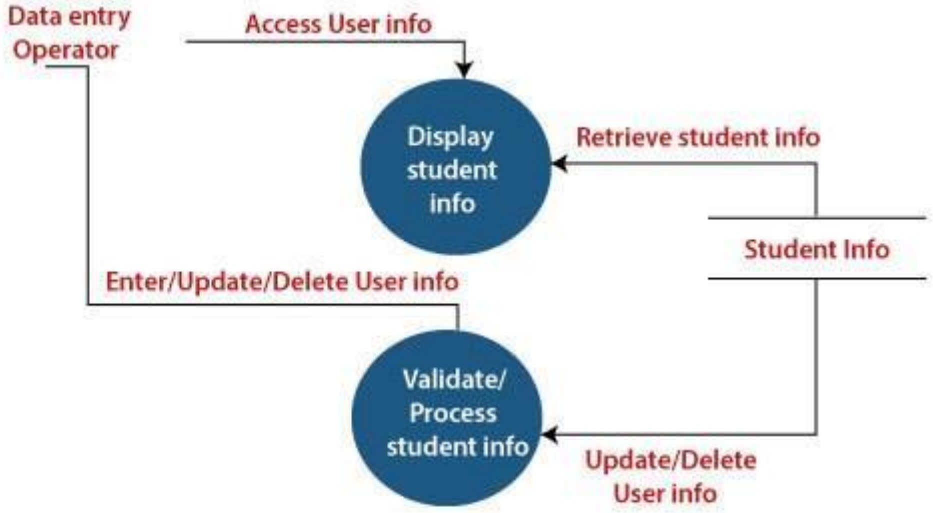


2. Login

The level 2 DFD of this process is given below:

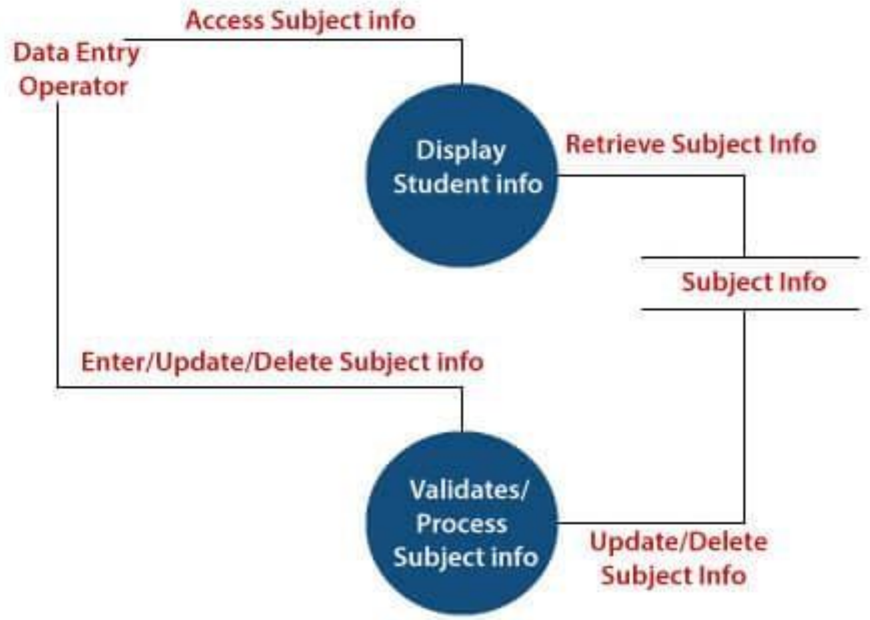

User Account info
Retrieve User info
Enter User ID, Password,Role
Data Entry Operator
Login
Enter User ID, Password,Role
Coordinator
Enter User ID, Password,Role
Marks Entry Clerk
Enter User ID, Password, Role
Administrator

3. Student Information Management



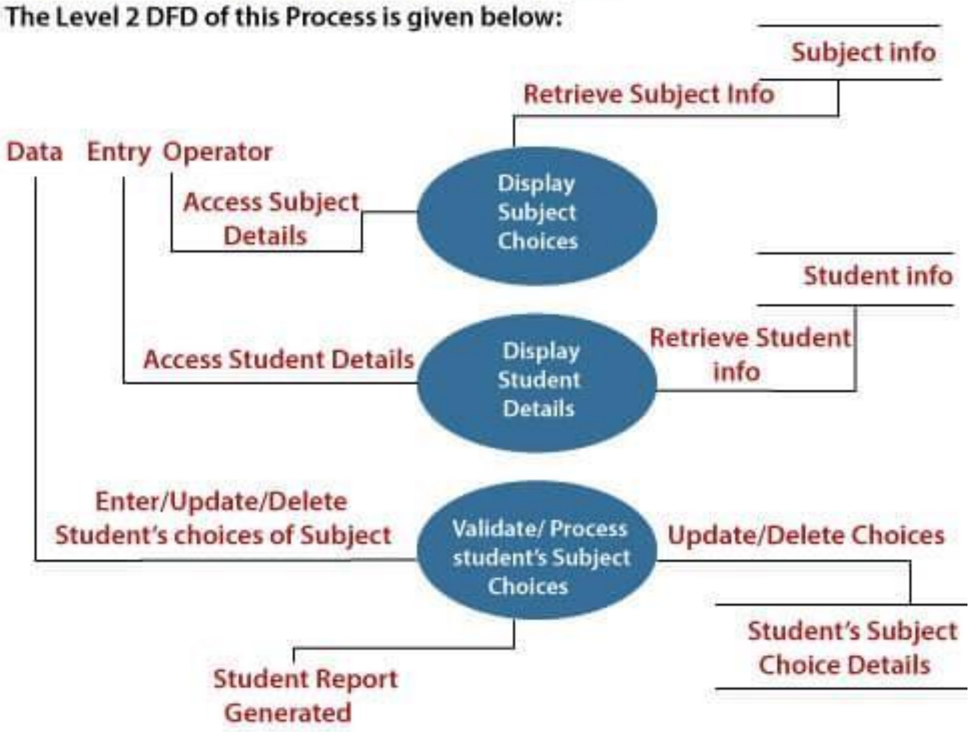
4. Subject Information Management

The level 2 DFD of this process is given below:



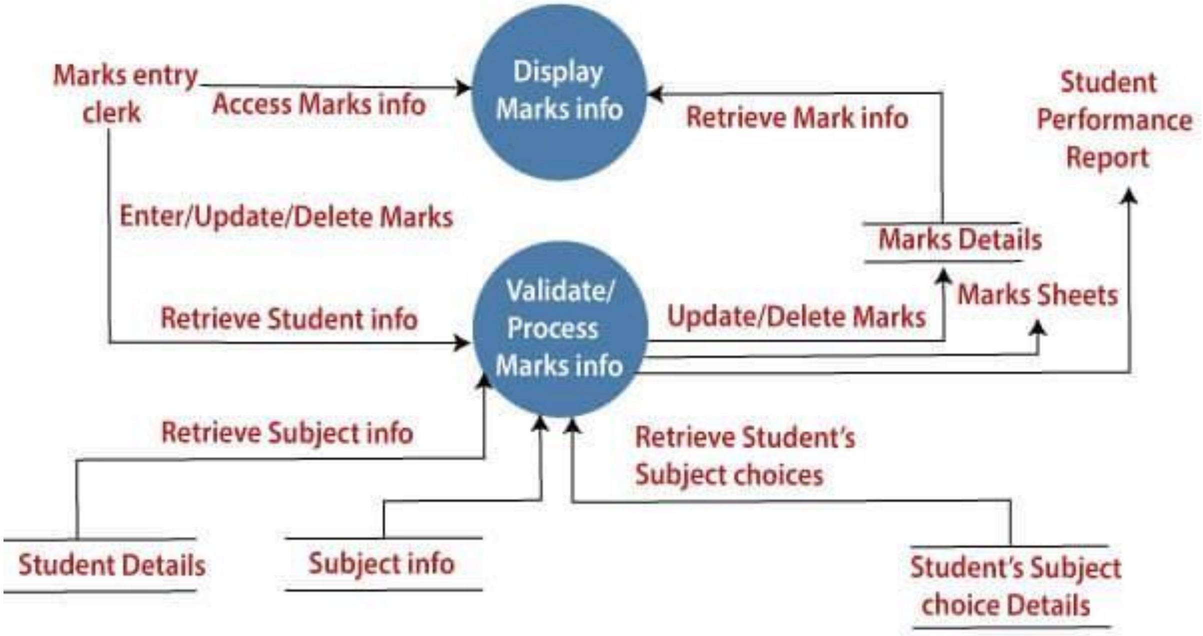
5. Student's Subject Choice Management

The Level 2 DFD of this Process is given below:



6. Marks Information Managment

The Level 2 DFD of this Process is given below:



Transaction Control Transformation

A Transaction Control transformation is an active and connected transformation. It allows us to commit and rollback transactions based on a set of rows that pass through a Transaction Control transformation.

Commit and rollback operations are of significant importance as it guarantees the availability of data.

A transaction is the set of rows bound by commit or rollback rows. We can define a transaction based on the varying number of input rows. We can also identify transactions based on a group of rows ordered on a common key, such as employee ID or order entry date.

When processing a high volume of data, there can be a situation to commit the data to the target. If a commit is performed too quickly, then it will be an overhead to the system.

If a commit is performed too late, then in the case of failure, there are chances of losing the data. So the Transaction control transformation provides flexibility.

In PowerCenter, the transaction control transformation is defined in the following levels, such as:

- **Within a mapping:** Within a mapping, we use the Transaction Control transformation to determine a transaction. We define transactions using an expression in a Transaction Control transformation. We can choose to commit, rollback, or continue on the basis of the return value of the expression without any transaction change.
- **Within a session:** We configure a session for the user-defined commit. If the Integration Service fails to transform or write any row to the target, then We can choose to commit or rollback a transaction.

When we run the session, then the Integration Service evaluates the expression for each row that enters the transformation. When it evaluates a committed row, then it commits all rows in the transaction to the target or targets. When the Integration Service evaluates a rollback row, then it rolls back all rows in the transaction from the target or targets.

If the mapping has a flat-file as the target, then the integration service can generate an output file for a new transaction each time. We can dynamically name the target flat files. Here is the example of creating flat files dynamically - Dynamic flat-file creation.

TCL COMMIT & ROLLBACK Commands

There are five in-built variables available in the transaction control transformation to handle the operation.

1. **TC_CONTINUE_TRANSACTION**

   The Integration Service does not perform any transaction change for the row. This is the default value of the expression.

2. **TC_COMMIT_BEFORE**

   The Integration Service commits the transaction, begins a new transaction, and writes the current row to the target. The current row is in the new transaction.

   In tc_commit_before, when this flag is found set, then a commit is performed before the processing of the current row.

3. **TC_COMMIT_AFTER**

The Integration Service writes the current row to the target, commits the transaction, and begins a new transaction. The current row is in the committed transaction.

In tc_commit_after, the current row is processed then a commit is performed.

4. **TC_ROLLBACK_BEFORE**

   The Integration Service rolls back the current transaction, begins a new transaction, and writes the current row to the target. The current row is in the new transaction.

   In tc_rollback_before, rollback is performed first, and then data is processed to write.

5. **TC_ROLLBACK_AFTER**

   The Integration Service writes the current row to the target, rollback the transaction, and begins a new transaction. The current row is in the rolled-back transaction.

   In tc_rollback_after data is processed, then the rollback is performed.

How to Create Transaction Control Transformation

Follows the following steps to create transaction control transformation, such as:

Step 1: Go to the mapping designer.

Step 2: Click on transformation in the toolbar, and click on the **Create** button.

Step 3: Select the transaction control transformation.

Step 4: Then, enter the name and click on the **Create** button.

Step 5: Now click on the **Done** button.

Step 6: We can drag the ports into the transaction control transformation, or we can create the ports manually in the ports tab.

Step 7: Go to the properties tab.

Step 8: And enter the transaction control expression in the Transaction Control Condition.

Configuring Transaction Control Transformation

Here are the following components which can be configuring in the transaction control transformation, such as:

1. **Transformation Tab**: It can rename the transformation and add a description.
2. **Ports Tab**: It can create input or output ports.
3. **Properties Tab**: It can define the transaction control expression and tracing level.
4. **Metadata Extensions Tab**: It can add metadata information.

Transaction Control Expression

We can enter the transaction control expression in the Transaction Control Condition option in the properties tab.

The transaction control expression uses the IIF function to check each row against the condition.

Syntax

Here is the following syntax for the Transaction Control transformation expression, such as:

1. IIF (condition, value1, value2)   For

example:

1. IIF (dept_id=11, TC_COMMIT_BEFORE,TC_ROLLBACK_BEFORE)

Example

In the following example, we will commit data to the target when dept no =10, and this condition is found true.

Step 1: Create a mapping with EMP as a source and EMP_TARGET as the target.

Step 2: Create a new transformation using the transformation menu, then

1. Select a transaction control as the new transformation.

2. Enter transformation name tc_commit_dept10.

3. And click on the create button.

Step 3: The transaction control transformation will be created, then click on the done button.

Step 4: Drag and drop all the columns from source qualifier to the transaction control transformation then link all the columns from transaction control transformation to the target table.

Step 5: Double click on the transaction control transformation and then in the edit property window:

1. Select the property tab.
2. Click on the transaction control editor icon.

Step 6: In the expression editor enter the following expression:

1. "iif(deptno=10,tc_commit_before,tc_continue_transaction)".
2. And click on the OK button.
3. It means if deptno 10 is found, then commit transaction in target, else continue the current processing.

Step 7: Click on the OK button in the previous window.

Now save the mapping and execute it after creating sessions and workflows. When the department number 10 is found in the data, then this mapping will commit the data to the target.

User Interface Design

Characteristics of Good Interface

The ability of any website or a web application to attract and engage users ultimately depends on how well the user interface is designed. Even the best developed service or product will be a failure if the user finds it to be too complicated to navigate or too confusing and feels it is monotonous or unattractive. A good user interface allows the user to carry out the intended actions efficiently and effectively, without causing too much of a distraction. User interface is the only way you can communicate with your client accessing your site remotely.

A good UI should be able to achieve business goals while keeping in mind the requirements of the user providing excellent UX to the user. Providing good User Experience is invariably an important component of a good UI.

The points to be kept in mind while designing good user interface are:

1. **Clear and Simple** : A good user interface provides a clear understanding of what is happening behind the scenes or provides visibility to the functioning of the system. The whole purpose of user interface design is to enable the user to interact with your system by communicating meaning and function. Obviously, if the interface too complex to navigate, it might annoy the user and make him or her leave the page quickly and move on to some thing else. Make sure the interface is understandable and simple to navigate through.
2. **Creative but familiar** : When the users are familiar with something and know how it behaves, navigation becomes easier. In effect, the user expects to see what is familiar to him or her. It is good to identify things that your users are accustomed to and integrate them into your user interface. At the same time, users appreciate some thing creative, not so runoff-the-mill experience. But while being creative it should be kept in mind not to lose the familiarity component.
3. **Intuitive and consistent** : The controls and information must be laid out in an intuitive and consistent way for an interface to be easy to use and navigate. It's not good to drastically change the lay out to achieve the changing functionality the business may require from time to time. The design process should be based on the logic of usability -features that are the most frequently used should be the most prominent in the UI and controls should be made consistent so that users know how to repeat their action.
4. **Responsive** : If the interface fails to keep up with the demands of the user, this will significantly diminish their experience and can result in frustration, particularly when trying to perform basic tasks. Wherever possible, the interface should move swiftly in pace with the user. Being responsive means being fast. The interface, if not the program behind it, should work fast. Waiting for things to load might make the user frustrated.
5. **Maintainable** : A UI should have the capacity for and changes to be integrated without causing a conflict of interest. For instance, you may need to add an additional feature to the software, if your interface is so convoluted that there is no space to draw attention to this feature without compromising something else or appearing unaesthetic, then this signifies a flaw in design.

The better the user interface the easier it is to guide people to use it and it also reduces the training costs. The better your user interface the less help people will require exploiting it and keep them coming back.

Basic Components of UID

User Interface Design

User interface design is also known as user interface engineering. User interface design means the process of designing user interfaces for software and machines like a mobile device, home appliances, computer, and another electronic device with the aim of increasing usability and improving the user experience.

The aim of user interface design is to make user experiences as easy as possible while still being successful in achieving user goals (user-centered design).

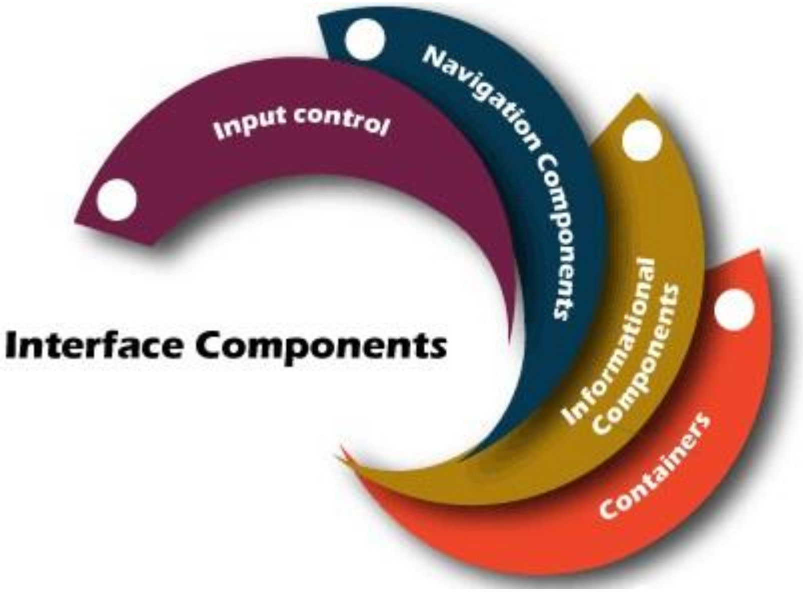
A well-designed user interface design makes it easier to complete the task in hand without drawing needless attention to itself. Graphic design and typography are used to influence its utility by influencing how the consumer interacts with it and improving the design's aesthetic appeal. Design aesthetics can increase or decrease the user's ability to use interface's functions. The design process should balance technical functionality as well as visual elements (for example, mental model) in order to build a system that is not only operational but as well usable and adaptable to evolving user requirements.

Interface design is used in various projects, including computer systems, commercial planes, automobiles; many of these projects include many of the same basic human interactions, but they often include certain special skills and experience. Consequently, whether it is software design, industrial design, user research or web design, designers prefer to specialize in those types of projects and have skills- based around their experience.

Choosing Interface Components

Users have become aware of interface components acting in a certain manner, so try to be predictable and consistent in our selections and their layout. As a result, task completion, satisfaction, and performance, will increase.

Interface components may involve:



1. Input controls
2. Navigational Components
3. Informational Components
4. Containers

Input Controls: Input Controls involve buttons, toggles, dropdown lists, checkboxes, date fields, radio buttons, and text fields.

Navigational Components: Navigational components contain slider, tags, pagination, search field, breadcrumb, icons.

Informational Components: Informational Components contain tooltips, modal windows, progress bar, icons, notification message boxes.

Containers: Containers include accordion.

Many components may be suitable to display content at times. When this happens, it is crucial to think about this trade-off. For example, sometimes, components that may help you space, place more focus on the user, forcing them to guess what the dropdown is or what the element might be.

Best Practices for Designing an Interface

It All starts with getting to know your users, which contains understanding about their interests, abilities, tendencies, and habits. If you have figured out who your customer is, keep the following in mind when designing your interface: ○ Create consistently and use common UI components

○ Use typography to make hierarchy and clarity.

○ Make sure that the system communicates what's happening

○ Use color and texture strategically ○ Keep the interface simple ○ Be purposeful in page layout

Create Consistently and Use Common UI Components

Users would feel more at ease and be able to complete tasks more easily if we use common components in our UI. It's also important to generate pattern in language, design, and layout across the website in order to help with productivity. If a user has mastered one ability, they should be able to apply it to others areas of the website.

Use Typography in Order to Make Hierarchy and Clarity

Think about how we are going to use the typeface. Text in various sizes, fonts, and arrangements in order to help increase readability, legibility, and scanability.

Make Sure that the System Communicates What's Happening

Always keep your user up to date on their change in state, location, errors, actions, etc. Using various UI components to communicate status and, if needed, the next steps will help your user feel less frustrated.

Use color and Texture Strategically

Using contrast, light, color, and texture to our benefit, we can draw attention to or draw attention away from objects.

Keep the Interface Simple

Mostly the great interfaces are not visible to the user. They avoid needless components and use simple terminology on labels and in messaging.

Be Purposeful in Page Layout

Take into account the spatial associations between the objects on the page and organize the page on the basis of importance. Carefully positioning objects can aid scanning and readability by drawing attention to the most appropriate pieces of information.

Designing User Interfaces for Users

User interfaces are the points of interaction between the user and developer. They come in three different types of formats:

1. Graphical User Interfaces (GUIs)
2. Gesture-based Interfaces
3. Voice-controlled Interfaces (VUIs)

1. Graphical User Interface (GUIs)

In the Graphical user interface, the users can interact with visual representations on the digital control panels. Example of GUI, a computer's desktop.

2. Gesture-Based Interfaces

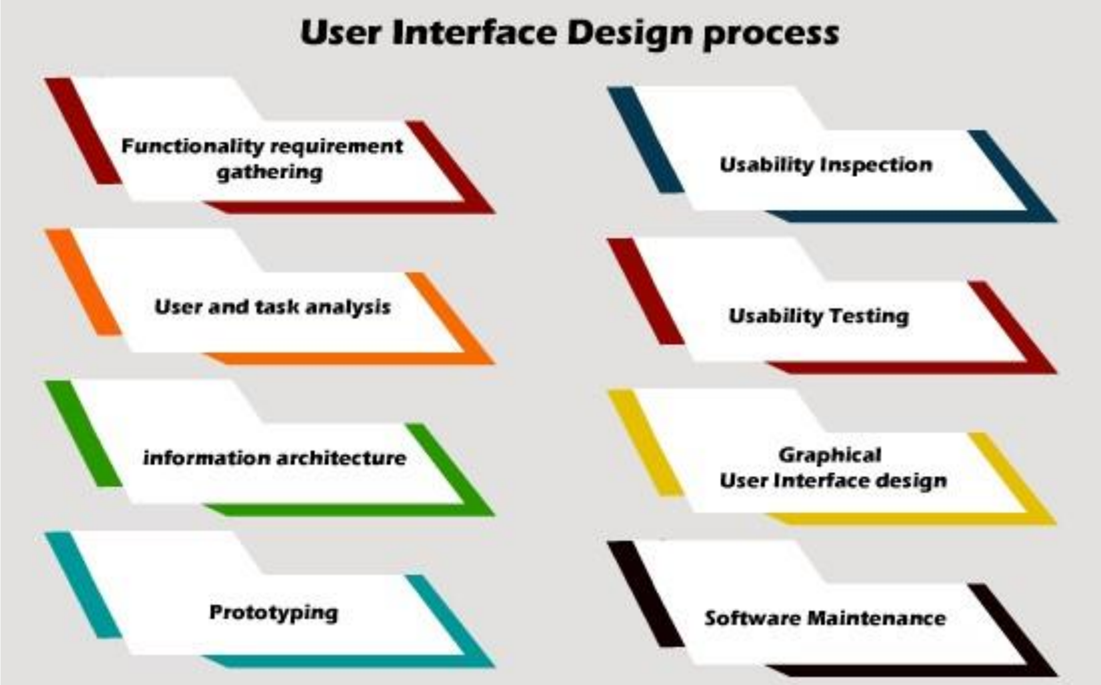
In gesture-based interfaces, users can interact with 3D design spaces by moving their bodies. Example of Gesture-Based Interfaces, Virtual Reality (VR) games.

3. Voice-Controlled Interfaces (VUIs)

In, Voice-controlled interfaces (VUIs), users can interact with the help of the voice. Example of Voice-Controlled Interfaces (VUIs), Alexa on Amazon devices, and Siri on iPhone.

User Interface Design Processes

The user-interface design necessitates an in-depth understanding of user requirements. It primarily focuses on the platform's requirements and user preferences. There are several stages and procedures of user interface design, some of which are more demanding than others depending on the project.



Functionality Requirements Gathering

Creates a list of device functionalities that are needed to fulfil the user's project goal and specification.

User and Task Analysis

It is the kind of field research. It is the research of how the system's potential users perform the tasks that the design would serve, and perform interviews to learn more about their goals.

Typical questions involve:

- What do you think the user would like the system to do?
- What role does the system fit in the user's everyday activities or workflow?
- How technically savvy is the user, and what other systems does the user already use?
- What styles of user interface look and feel do you think the user prefers?

Information Architecture

Process development or the system's information flow (means for phone tree systems, this will be a choice tree flowchart for phone tree systems, and for the website, this will be site flow that displays the page's hierarchy).

Prototyping

The wire-frame's the development either in the form of simple interactive screens or paper prototypes. To focus on the interface, these prototypes are stripped of all look and feel components as well as the majority of the content.

Usability Inspection

Allowing an evaluator to examine a user interface. It is typically less expensive to implement as compared to usability testing, and in the development process, it can be used early. It may be used early in the development process to determine requirements for the system, which are usually unable to be tested on the users. There are various usability inspection procedures such as a cognitive walkthrough, which focuses on how easy it is for new users to complete tasks with the system for new users, pluralistic walkthrough, which involves a group of people step through a task scenario and discussing usability issues, heuristic evaluation, that uses a series of heuristic to find usability issues in the UI design.

Usability Testing

Prototypes are tested on a real user, often using a method known as think-aloud protocol, in which we can ask the user to speak about their views during the experience. The testing of user interface design permits the designer to understand the reception from the viewer's perspective, making it easier to create effective applications.

Graphical User Interface Design

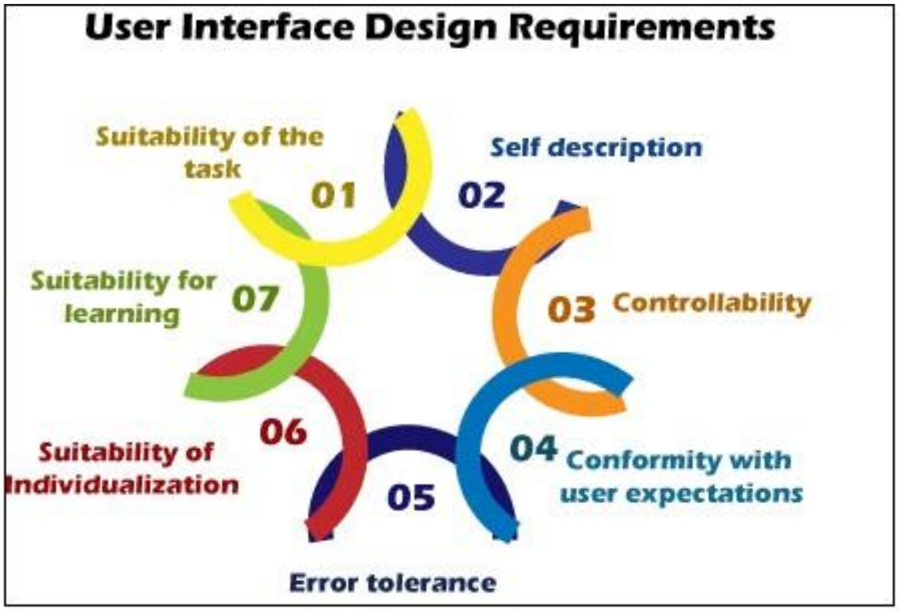
It is the actual look and feel of the design of the final graphical user interface (GUI). These are the control panels and faces of design; voice-controlled interfaces contain oralauditory interaction, while gesture-based interfaces users involve with 3D design spaces through physical motions. This can be based on findings developed during user research and refined to correct and usability problems found via the testing's results. This process typically includes some computer programming in order to validate forms, establish links, or perform a desired action, depending on the type of interface being developed.

Software Maintenance

After a new interface is deployed, it may be necessary to perform routine maintenance in order to fix software bugs, add new functionality or fully update the system. When the decision is taken to update the interface, the legacy system will go through a new iteration of the design process. The stages of the interface life cycle will continue to repeat.

User Interface Design Requirements

The dynamic characteristics of a system are defined in terms of the dialogue requirements contained in 7 principles of part 10 of the ergonomics standard, the ISO 9241. This standard provides a system of ergonomic "principles" for the dialogue techniques along with the high-level concepts, examples, and implementations. The principles of the dialogue reflect the interface's dynamic aspects and mostly thought of as the interface's "feel." The following are the seven dialogue principles:



1. Suitability of the Task

The dialogue is appropriate for the task when it helps the user in completing the task efficiently and effectively.

2. Self-Descriptiveness

When each dialogue phase is instantly understandable due to system feedback or clarified to the user upon request, the dialogue is self-descriptive.

3. Controllability

When the user is capable to initiate and monitor the course and speed of the interaction until the aim is achieved, then dialogue is controllable.

4. Conformity with User Expectations

If the dialogue is reliable and corresponds to the characteristics of the user, like experience, education, task awareness, and generally accepted conventions, it conforms to user experience.

5. Error Tolerance

If, despite obvious errors in input, the desired outcome can be accomplished with no or limited action from the user, then the dialogue is error-tolerant.

6. Suitability for Individualization

If the interface software can be changed to meet the job needs, individual interests, and abilities of the user, the dialogue is able of individualization.

7. Suitability for Leaning

The dialogue support for learning as it assists and guides the user in learning how to use the system.

The ISO 9241 standard defines usability as the effective performance and the satisfaction of the consumer. The following is an explanation of usability found in Part 11.

- The degree to which the overall system's expected objectives of use are met is how usable it is (effectiveness).
- The resources must be spent in order to achieve the desired outcomes (efficiency). o The degree to which the user finds the entire system acceptable (satisfaction).

Usability factors include effectiveness, efficiency, and satisfaction. In order to assess these factors, they must first be split into sub-factors and then into usability measures.

Part 12 of the ISO 9241 standard specifies the organization of information such as alignment, arrangement, location, grouping, arrangement, display of the graphical objects, and the information's coding (colour, shape, visual cues, size, abbreviation) by seven attributes. The attributes of the presented information reflect the interface's static aspects and can be referred to as the interface's "look." The attributes are defined in detail in the standard's recommendations. Each of the seven qualities is supported by one or more of the recommendations. The seven-presentation characteristic are as follows: o **Clarity:** - The information content is conveyed easily and correctly.

- **Discriminability:** - The displayed data can be separated with precision. o **Conciseness:** - The users are not overburdened with irrelevant data.
- **Consistency:** - Consistency means a unique design with conformity with the expectation of users. o **Detectability:** - The attention of the user is directed towards the essential information essential.
- **Legibility:** - Legibility means information is easy to read.
- **Comprehensibility:** - The meaning is straightforward, recognizable, unambiguous, and easy to comprehend.

The user guidance in part 13 of the ISO 9241 standard states that it should be easily distinguishable from other shown information and must be precise for the use of present context. The following five methods can be used to provide user guidance:

- Prompts indicating that the system is available for input explicitly (specific prompts) or implicitly (generic prompts).
- Feedback informing related to the input of the user timely, non-intrusive, and perceptible.

- Details about the application's current state, the system's hardware and software, and the user's activities.
- Error management contains error detection, error correction, error message, and user support for error management.
- Online assistance for both system-initiated and user-initiated requests with detailed information for the current context of usage.

How to Make Great UIs

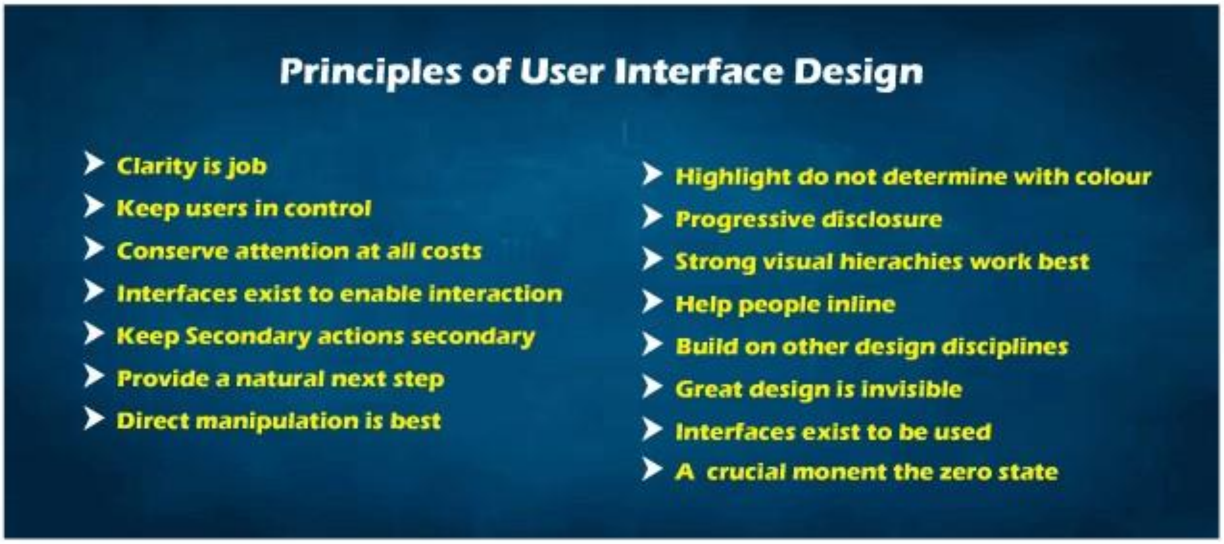
Remember that the users are people with needs like comfort and a mental capacity limit when creating a stunning GUI. The following guidelines should be followed:

1. Create buttons, and other popular components that behave predictably (with responses like pinch-to-zoom) so that users can use them without thinking. Form must follow function.
2. Keep your discoverability high. Mark icons clearly and well-defined affordances, such as shadows for buttons.
3. The interface should be simple (including elements that help users achieve their goals) and create an "invisible" feel.
4. In terms of layout, respect the user's eyes and attention. Place emphasis on hierarchy and readability:

    - ***User proper alignment:*** Usually select edge (over center) alignment.
    - ***Draw attention to Key features using:***
    - Colour, brightness, and contrast are all important factors to consider Excessive use of colors or buttons should be avoided.
    - Font sizes, italics, capitals, bold type/weighting, and letter spacing are all used to create text. User should be able to Deduce meaning simply by scanning.
    - Regardless of the context, always have the next steps that the user can naturally deduce.
    - Use proper UI design patterns to assist users in navigating and reducing burdens such as pre-fill forms. Dark patterns like hard-to-see prefilled opt-in/opt-out checkboxes and sneaking objects into the user's carts should be avoided.
    - Keep user informed about system responses/actions with feedback.

Principles of User Interface Design

The following are the principles of user interface design:



Principles of User Interface Design

- Clarity is job
- Keep users in control
- Conserve attention at all costs
- Interfaces exist to enable interaction
- Keep Secondary actions secondary
- Provide a natural next step
- Direct manipulation is best
- Highlight do not determine with colour
- Progressive disclosure
- Strong visual hierachies work best
- Help people inline
- Build on other design disciplines
- Great design is invisible
- Interfaces exist to be used
- A crucial monent the zero state

1. Clarity is Job

The interface's first and most essential task is to provide clarity. To be effective in using the interface you designed, people need to identify what it is, regardless of why they will use it, understand what the interface is doing in interaction with them. It assists them in anticipating what will occur as they use it. And then effectively interact with it in order to be effective. In interface, there is a space for mystery ad delayed gratification, but not for uncertainty. Clarity instils trust and encourages continued use. One hundred uncluttered screens are superior to one cluttered screen.

2. Keep Users in Control

Humans are most at ease when they have control of themselves and their surroundings. Unthoughtful software robs people of their comfort by dragging them into unexpected encounters, unexpected outcomes, and confusing pathways. Maintain user control by surfacing system status regularly, explaining causation (what will happen if you do this), and providing insight into what to expect at each turn. Don't be concerned with stating the obvious... the obvious rarely is.

3. Conserve Attention at All Cost

We live in a world that is constantly interrupted. It is difficult to read in peace these days without anything attempting to divert our focus. Attention is a valuable commodity. Distracting content should not be strewn around the side of your applications... keep in mind why the screen exists in the first place. Allow someone to finish reading before displaying an advertisement if they are currently reading. If you pay attention, then your readers will be happier, and your performance will be higher. When the primary aim is to make something useful, paying attention is a must. Preserve it at all costs.

4. Interfaces Exist to Enable Interaction

Interaction between humans and our world is allowed by interfaces. They can support, explain, allow, display associations, illuminate, bring us together, separate us, handle expectations, and provide access to service. Designing a user interface is not an artistic endeavour. Interfaces are not stand-alone landmarks. Interfaces perform a function, and their efficiency can be calculated. However, they are not just utilitarian. The best user interfaces can encourage, mystify, evoke and deepen our connection with the world.

5. Keep Secondary Actions Secondary

Multiple secondary actions may be added to screens with a single primary action, but they must be held secondary. Your article presents not so that individuals can post it on Twitter but so that people can read and comprehend it. Secondary action should be secondary by giving them a lighter visual weight or displaying them after the primary action is completed.

6. Provide a Natural Next Step

Few interactions are intended to be the last, so consider designing the last move for every interaction used with your interface. Predict what the next interaction will be and design to accommodate it. Just as we are interested in human conversation, offer an opportunity for more discussion. Don't leave anyone hanging because they did what you wish them to do…. Provide them with a natural next move that will assist them in achieving their objectives.

7. Direct Manipulation is Best

There is no need for an interface if we can directly access the physical objects in our universe. We build interfaces to help us interact with objects because this is not always easy, and objects are becoming increasingly informational. It is simple to add extra layers than required to an interface, creating overly-wrought buttons, attachments, options, graphics, windows, preferences, chrome, and other cruft, causing us to manipulate the interface. Instead of focusing on what matters, UI components ae includes, rather go back to your target of direct manipulation…design an interface with the smallest possible footprint while recognizing as many natural human movements as possible. In an ideal world, the interface is so light that the user feels as though they are directly manipulating the object of their focus.

8. Highlight, Don't Determine, with Colour

When the light changes, the colour of the physical object changes. In the full light of day, we see very different tree outlines against a sunset. As in the real world, where colour is a multi-shaded object, colour does not decide anything in an interface. It can be useful for highlighting and directing focus, but it should not be the only way to distinguish objects. Using light or muted background colours for prolonged screen time, saving brighter hues for accents. Of course, there is a time and place for bright or vibrant background colours; just make sure they are suitable for the target audience.

9. Progressive Disclosure

On each screen, just show what is needed. If people must make a decision, give them sufficient information in order to make that decision, then go into more details on a subsequent screen. Avoid the popular trap of over-explaining or showing all at once. Defer decisions to subsequent screens wherever possible by gradually revealing information as needed. Your experiences would be clearer as a result of this.

10. Strong Visual Hierarchies Work Best

When the visual elements on a computer are arranged in a simple viewing order, it creates a powerful visual hierarchy. This means when users consistently see the same objects in the same order. The weak visual hierarchies offer some guidance related to where one should gaze and relax and feel disorganized and confused. It is difficult to maintain a clear visual hierarchy in fast-changing environments because visual weight is relative; if nothing is bold or everything is bold. If a single visually heavy element is included in a screen, then the designer has to reset the visual weight of all other elements in order to achieve a strong hierarchy once more.

11. Help People Inline

Help is not needed in ideal interaction because the interface is usable and learner. The step below that, fact, is one in which assistance is inline and contextual, accessible only when and where it is required and concealed at all other times. When you ask people to go help and find an answer to their question, you are putting the responsibility on them to understand what they want, rather than incorporate assistance where it is needed. Only make sure it is not in the way of people who are already familiar with your interface.

12. Build on Other Design Principle

Visual and graphic design, visualization, typography, information architecture, and copywriting all of these disciplines are the part of the interface design. They may be briefly discussed or trained in. Don't get caught up in turf battles or dismiss other disciplines; instead, take what you need from them and keep moving forward. Incorporate ideas from apparently unrelated fields as well... what can we learn from bookbinding, publishing code, skateboarding, karate, firefighting?

13. Great Design is Invisible

The interesting thing about good design is that it usually goes unobserved by the people who use it. One reason for this is that if the design is effective, then the user will be able to concentrate on their own objectives rather than the interface...They are happy when they achieve their goal and do not essentially reflect on the condition. As a designer, this can be difficult since we receive less praise when our work is successful. Great designers, on the other hand, are comfortable with a well-used design and understand that satisfied users are always silent.

14. Interfaces Exist to be Used

Interface design, like most design disciplines, is effective when people use what you have created. Design fails if people choose not to utilize it, just like a beautiful chair which is painful to sit in. As a result, interface design can be more related to building a userfriendly experience as it is about designing a useful artifact. It is not sufficient for an interface to fulfil the designer's ego: it has to be used!

15. A Crucial Moment: The Zero State

The first time a user interacts with an interface is critical, but designers often ignore it. It's better to plan for the zero state, or the state where nothing has happened yet, to great support our users get up to speed with our designs. This is not supposed to be a blank canvas...it should give you direction and point you in the right direction for getting up to speed. The initial context is where much of the friction of contact occurs...people have a much better chance of succeeding once they grasp the rules.

Mistakes to Avoid in UI Design

The following are the mistakes that we have to avoid in UI design:

- **Do not implement a user-centred design**. This is an easy part to overlook, but it is one of the most critical aspects of the UI design. User's desires, expectations, and the problems should all be considered when designing. Avoid doing, so it may have a negative effect on your company and lead to its demise.
- **Excessive use of dynamic effects:** Using a lot of animation effects is not always a sign of a good design. As a result, limiting the use of decorative animations will help to improve the user experience.
- **Preparing so much in advance:** Particularly in the early stages of design, we just need to have the appropriate image of the design in our heads and get to work. However, this

strategy is not always successful. At times, exploring other sources can show some unexpected results.

- **Not Leaning more about the target audience:** - This point once again, demonstrates what we have just discussed. Rather than designing with your own desires and taste in mind, imagine yourself as the consumer. Simply consider what the consumer will enjoy, and if possible, conduct an interview or survey some potential customers to get a better understanding of their requirements.

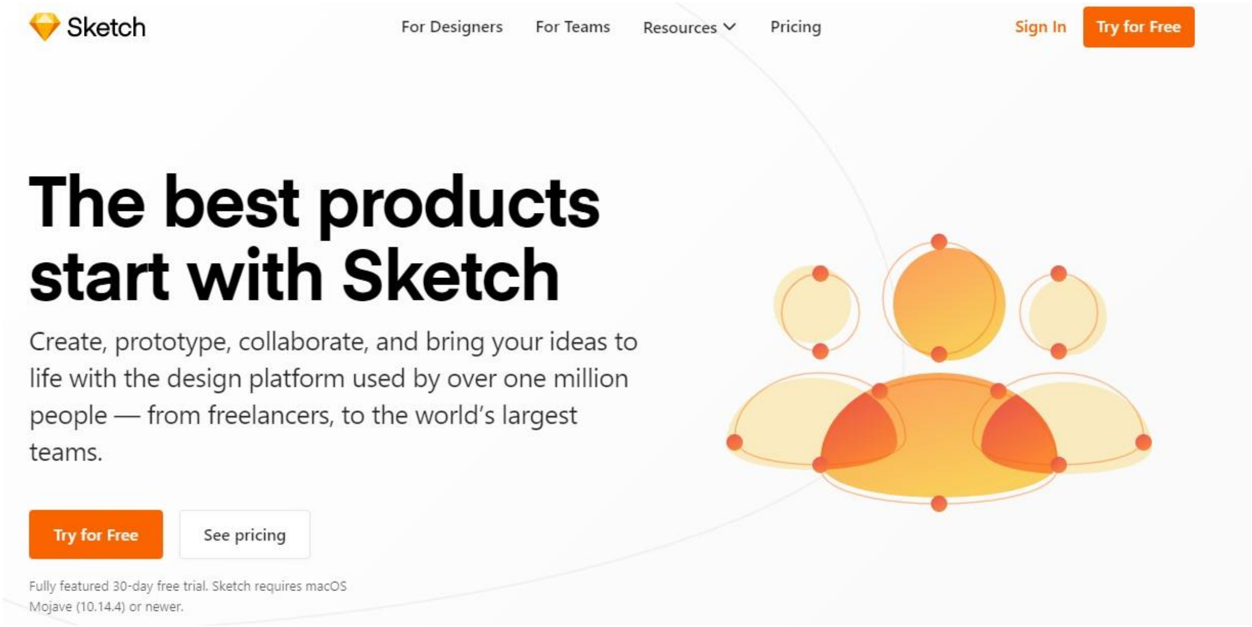
Essential Tools for User Interface Design

There are various essential tools for user interface design:

1. Sketch
2. Adobe XD
3. Invision Studios
4. UXPin
5. Framer X

1. Sketch

It is a user design tool mainly used by numerous UI and UX designers to design and prototyping mobile and web applications. The Sketch is a vector graphics editor that permits designers to create user interfaces efficiently and quickly.



The best products start with Sketch

Create, prototype, collaborate, and bring your ideas to life with the design platform used by over one million people — from freelancers, to the world's largest teams.

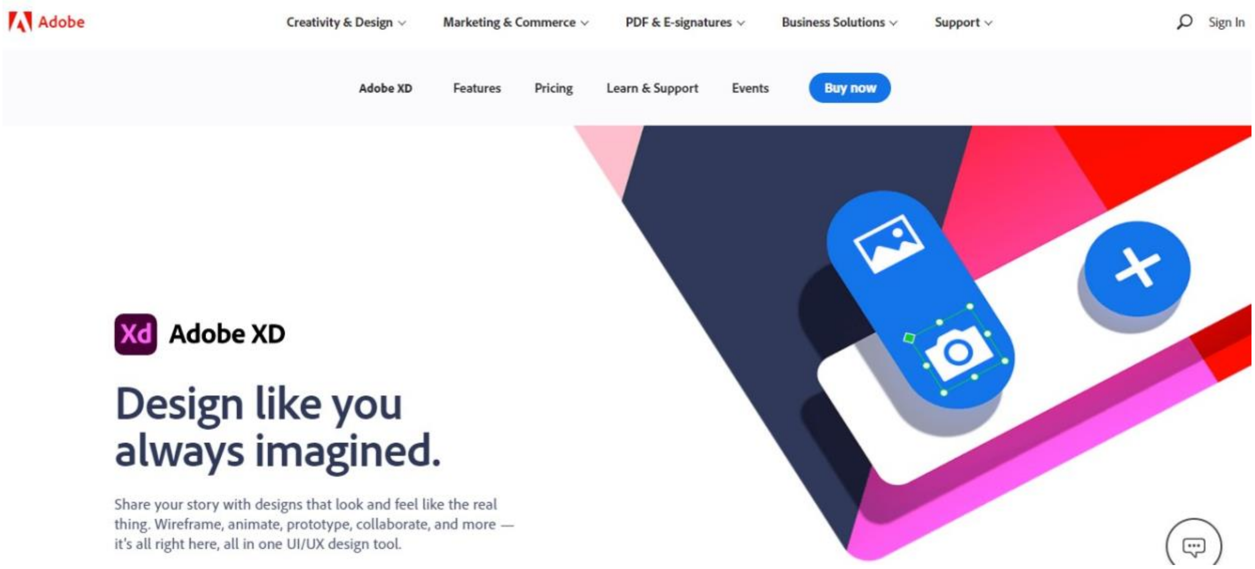
Fully featured 30-day free trial. Sketch requires macOS Mojave (10.14.4) or newer.

There are various features of Sketch:

- **Slicing and Exporting**
  Sketch gives users a lot of slicing control, allowing them to choose slice, and export any layer or object they want.
- **Symbols**
  Using this feature, user can build pre-designed elements which can be easily re-used as well as replicated in any artboard or project. This feature will help designers save time and build a design library for potential projects.
- **Plugins**
  Maybe a feature you are looking for is not available in the default sketch app. In that situation, you don't have to worry; there are number of created plugins that can be downloaded externally and added to our sketch app. The options are limitless!

2. Adobe XD

It is a vector-based tool. We use this tool for designing interfaces and prototyping for mobile applications as well as the web. Adobe XD is just like Photoshop and illustrator, but it focuses on user interface design. Adobe XD has the advantage of including UI kits for Windows, Apple, and Google Material Design, which helps designers create user interfaces for each device.



Design like you always imagined.

Share your story with designs that look and feel like the real thing. Wireframe, animate, prototype, collaborate, and more — it's all right here, all in one UI/UX design tool.

Features of Adobe XD

There are various features of Adobe XD:

- **Voice Trigger**

Voice Trigger is an innovative feature introduced by Adobe XD which permits prototypes to be manipulated via voice commands.

- **Responsive Resize**

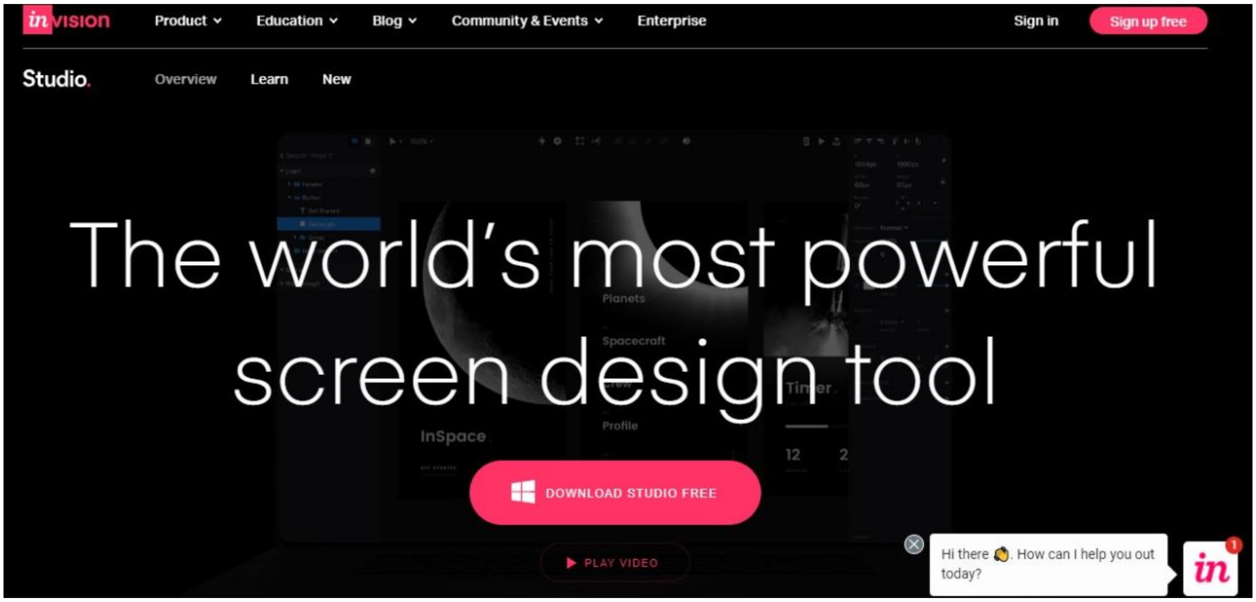
Using this feature, we can automatically adjust and resize objects/elements which are present on the artboards based on the size of the screen or platform required.

- **Collaboration**

We can connect Adobe XD to other tools like Slack, allowing the team to collaborate across platforms like Windows and macOS.

3. Invision Studios

It is a simple vector-based drawing tool with design, animation, and prototyping capabilities. Invision studios is a relatively new tool, but it has ready demonstrated a high level of ambition through its numerous available functionalities and remarkable prototyping capabilities. The ability to move and open files from sketch to Invision is an added benefit, allowing you to create more immersive user interfaces than you could with sketch alone.



Hi there 👋. How can I help you out today?

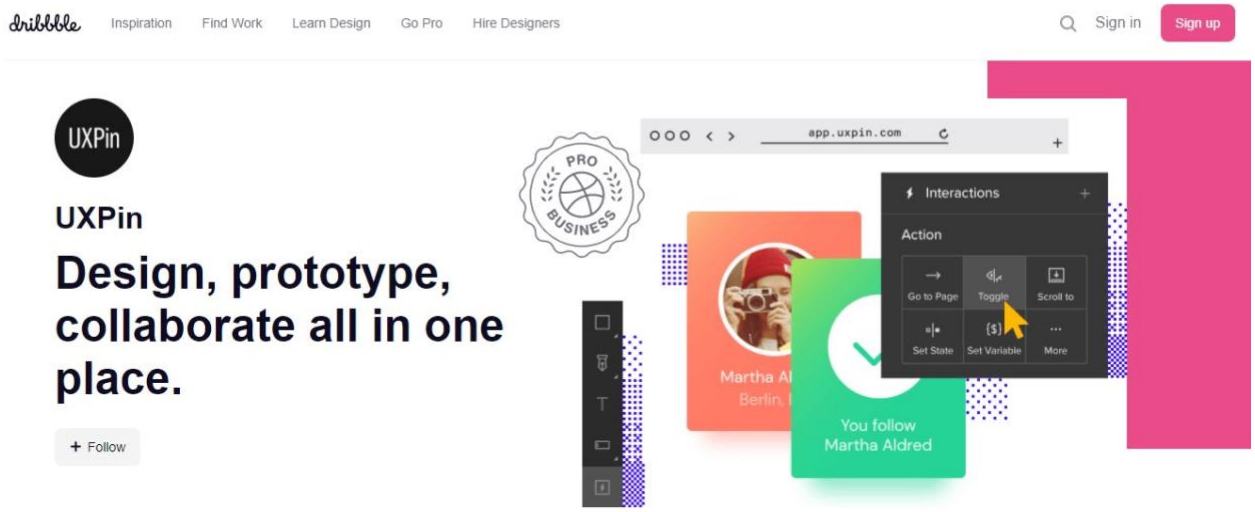
Features of the Invision Studios

There are various features of Invision studios:

- **Advanced Animations**
  With the various animations provided by studios, animating your prototype has become even more exciting. We can expect higher fidelity prototypes with this feature, including auto-layer linking, timeline editions, and smart-swipe gestures.
- **Responsive Design**
  The responsive design feature saves a lot of time because it eliminates the need of multiple artboards when designing for numerous devices. Invision studios permit users to create a single artboard that can be adjusted based on the intended device.
- **Synced Workflow**
  Studios enable a synchronised workflow across all projects, from start to finish, in order to support team collaboration. This involves real-time changes and live concept collaboration, as well as the ability to provide instant feedback.

4. UXPin

Another amazing tool for the design user interface is UXPin that comes with the abilities of designing and prototyping. In contrast to other user interface tools, this tool is recommended to be a better fit for large design teams and projects. UXPin also comes with UI element libraries which give you access to Material Design, iOS libraries, Bootstrap, and variety of icons.



UXPin

Design, prototype, collaborate all in one place.

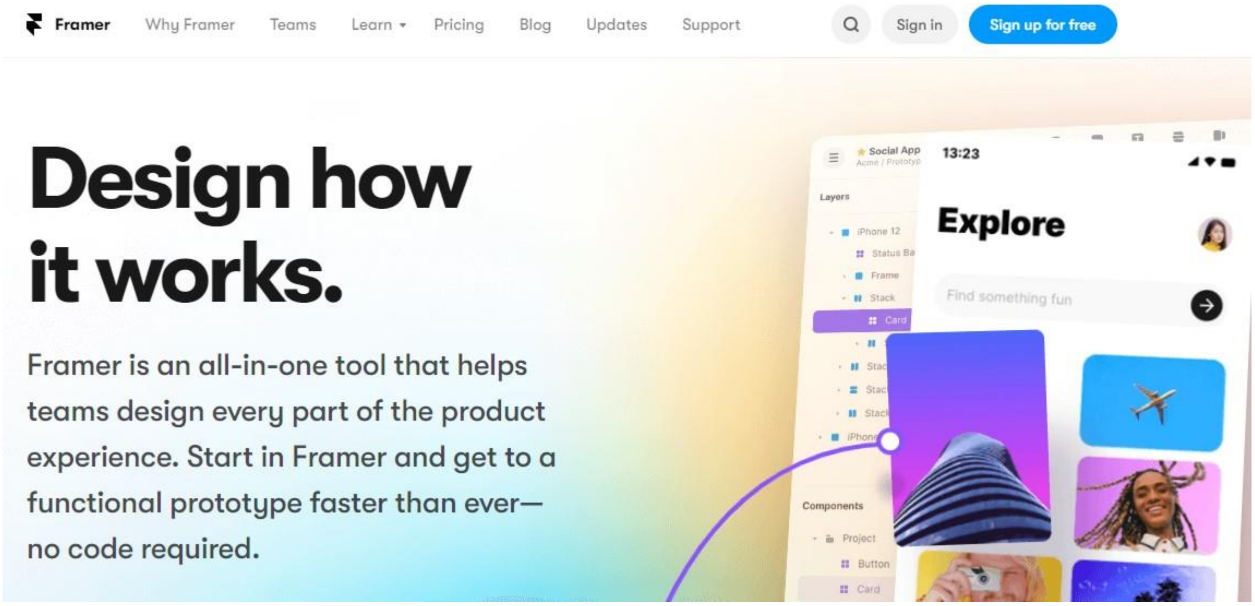
Features of UXPin

There are various features of UXPin:

○ Mobile support ○ Collaboration ○ Presentation tools ○ Drag and Drop ○ Mockup Creation ○ Protype Creation ○ Interactive Elements ○ Feedback Collection ○ Feedback Management

5. Framer X

Framer X was released in 2018. It is one of the newest design tools which is used to design digital products from mobile applications to websites. The interesting feature of this tool is the capability to prototype along with the advanced interactions and animations while also integrating the code's components. The React.js users feel that they are able to design and code on the same platform. Furthermore, Framer X allows users to build highly realistic prototypes that can be used to show clients or stakeholders the final product.



Design how it works.

Framer is an all-in-one tool that helps teams design every part of the product experience. Start in Framer and get to a functional prototype faster than ever—no code required.

Features of the Framer X

The following are the features of the Framer X:

○ From mockup to prototype, all in one
canvas ○ Framer X better support all types
of web fonts ○ Pixel-perfect designs with
rulers and guides ○ Get creative with precise
color management

Types of User Interface

The various types of user interfaces include:

- graphical user interface (GUI)
- command line interface (CLI)
- menu-driven user interface
- touch user interface
- voice user interface (VUI)
- form-based user interface
- natural language user interface

Components based GUI development

Component-based development (CBD) is a procedure that accentuates the design and development of computerbased systems with the help of reusable software components. With CBD, the focus shifts from software programming to software system composing.

Component-based development techniques involve procedures for developing software systems by choosing ideal off-the-shelf components and then assembling them using a well-defined software architecture. With the systematic reuse of coarse-grained components, CBD intends to deliver better quality and output.

Component-based development is also known as component-based software engineering (CBSE).

Software Coding & Testing

Coding

The coding is the process of transforming the design of a system into a computer language format. This coding phase of software development is concerned with software translating design specification into the source code. It is necessary to write source code & internal documentation so that conformance of the code to its specification can be easily verified.

Coding is done by the coder or programmers who are independent people than the designer. The goal is not to reduce the effort and cost of the coding phase, but to cut to the cost of a later stage. The cost of testing and maintenance can be significantly reduced with efficient coding.

Goals of Coding

1. **To translate the design of system into a computer language format:** The coding is the process of transforming the design of a system into a computer language format, which can be

executed by a computer and that perform tasks as specified by the design of operation during the design phase.

2. **To reduce the cost of later phases:** The cost of testing and maintenance can be significantly reduced with efficient coding.
3. **Making the program more readable:** Program should be easy to read and understand. It increases code understanding having readability and understandability as a clear objective of the coding activity can itself help in producing more maintainable software.

Code Review

Code Review is a systematic examination, which can find and remove the vulnerabilities in the code such as memory leaks and buffer overflows.

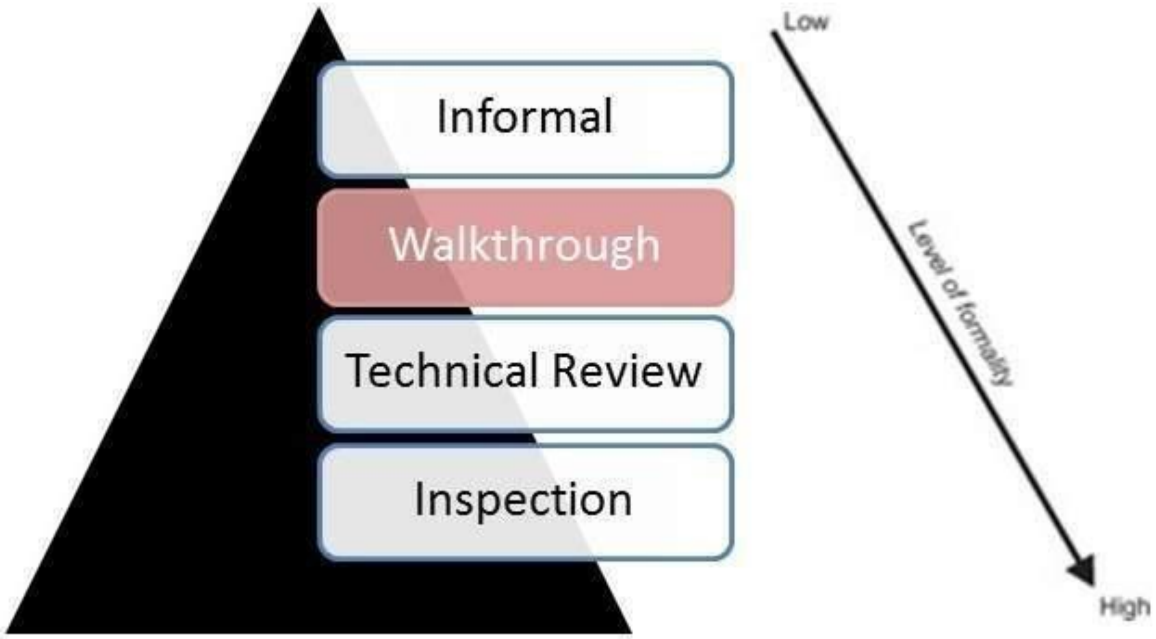
- Technical reviews are well documented and use a well-defined defect detection process that includes peers and technical experts.
- It is ideally led by a trained moderator, who is NOT the author.
- This kind of review is usually performed as a peer review without management participation.
- Reviewers prepare for the review meeting and prepare a review report with a list of findings.
- Technical reviews may be quite informal or very formal and can have a number of purposes but not limited to discussion, decision making, evaluation of alternatives, finding defects and solving technical problems.

Code walks through

Code Walkthrough is a form of peer review in which a programmer leads the review process and the other team members ask questions and spot possible errors against development standards and other issues.

- The meeting is usually led by the author of the document under review and attended by other members of the team.
- Review sessions may be formal or informal.
- Before the walkthrough meeting, the preparation by reviewers and then a review report with a list of findings.
- The scribe, who is not the author, marks the minutes of meeting and note down all the defects/issues so that it can be tracked to closure.
- The main purpose of walkthrough is to enable learning about the content of the document under review to help team members gain an understanding of the content of the document and also to find defects.

Where Code Walkthrough fits in ?



Code inspections and software Documentation

Code inspection in software engineering is the process of reviewing the code in an application to check for defects. Its purpose is to correct the issues in the programming language so the software performs at its highest potential. After the engineers build the product and write the code, they practice code inspection to find ways to minimize the time for the code to exercise commands. Resolving the issues that the inspection finds allows engineers to refine the internal structure of the software, as well as its security features.

In the software development process, software documentation is the information that describes the product to the people who develop, deploy and use it.

It includes the technical manuals and online material, such as online versions of manuals and help capabilities. The term is sometimes used to refer to source information about the product discussed in design documentation, code comments, white papers and session notes.

Software documentation is a way for engineers and programmers to describe their product and the process they used in creating it in formal writing. Early computer users were sometimes simply given the engineers' or programmers' notes. As software development became more complicated and formalized, technical writers and editors took over the documentation process.

Software documentation shows what the software developers did when creating the software and what IT staff and users must do when deploying and using it. Documentation is often incorporated into the software's user interface and also included as part of help documentation. The information is often divided into task categories, including the following:

- evaluating
- planning
- setting up or installing
- customizing
- administering
- using
- maintaining

Why is software documentation important?

Software documentation provides information about a software program for everyone involved in its creation, deployment and use. Documentation guides and records the development process. It also assists with basic tasks such as installation and troubleshooting.

Effective documentation gets users familiar with the software and makes them aware of its features. It can have a significant role in driving user acceptance. Documentation can also reduce the burden on support teams, because it gives users the power to troubleshoot issues.

Software documentation can be a living document that is updated over the software development lifecycle. Its use and the communication it encourages with users provides developers with information on problems users have with the software and what additional features they need. Developers can respond with software updates, improving customer satisfaction and user experience.

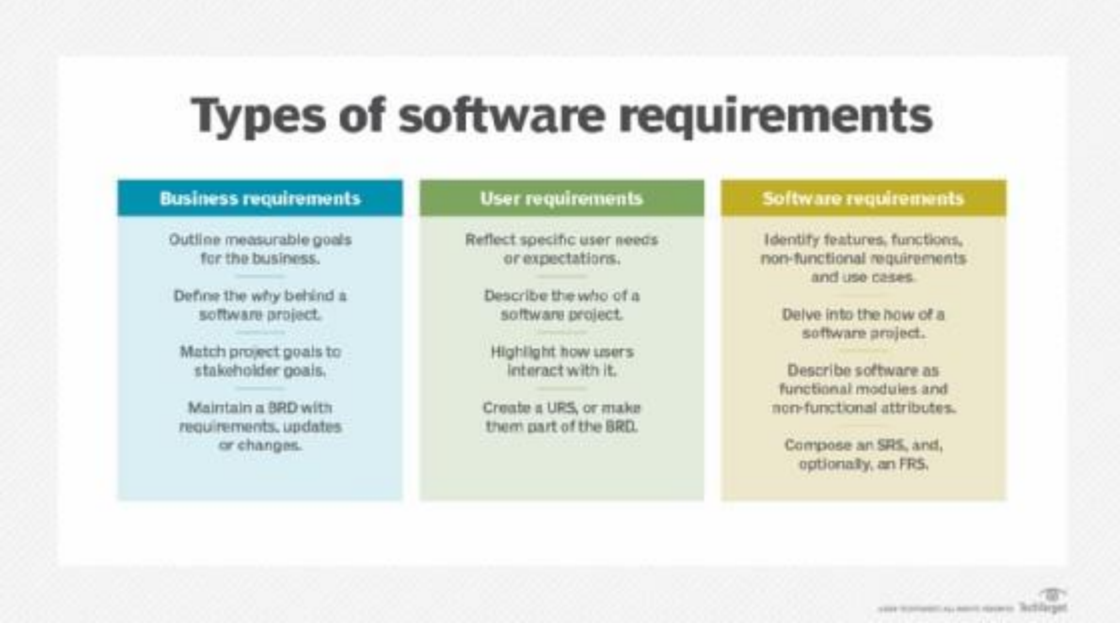
Types of software documentation

The two main types of software documentation are internal and external.

Internal software documentation

Developers and software engineers create internal documentation that is used inside a company. Internal documentation may include the following:

- **Administrative documentation.** This is the high-level administrative guidelines, roadmaps and product requirements for the software development team and project managers working on the software. It also may include status reports and meeting notes.
- **Developer documentation.** This provides instructions to developers for building the software and guides them through the development process. It includes requirements documentation, which describes how the software should perform when tested. It also includes architectural documentation that focuses on how all the components and features work together, and details data flows throughout the product.



Types of software requirements

Business requirements	User requirements	Software requirements
Outline measurable goals for the business.	Reflect specific user needs or expectations.	Identify features, functions, non-functional requirements and use cases.
Define the why behind a software project.	Describe the who of a software project.	Delve into the how of a software project.
Match project goals to stakeholder goals.	Highlight how users interact with it.	Describe software as functional modules and non-functional attributes.
Maintain a BRD with requirements, updates or changes.	Create a URS, or make them part of the BRD.	Compose an SRS, and, optionally, an FRS.

Software requirements are detailed in internal software documentation.

External software documentation

Software developers create this documentation to provide IT managers and end users with information on how to deploy and use the software. External documentation includes the following:

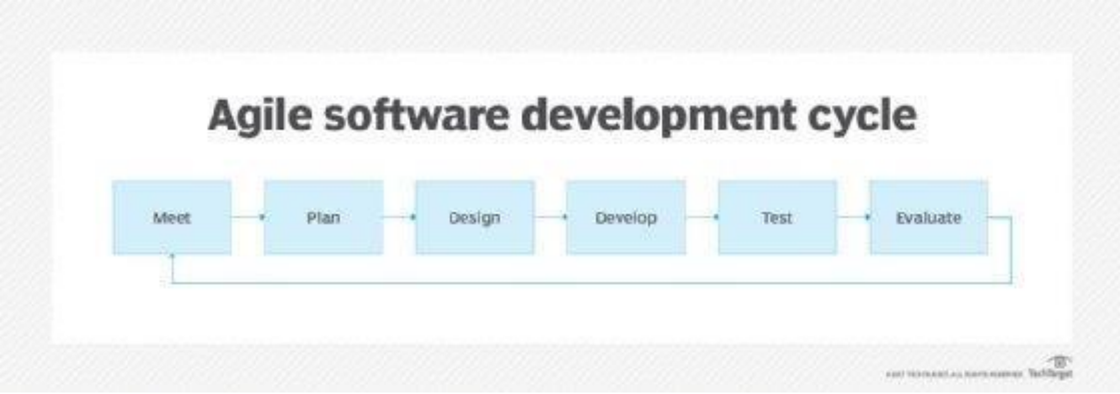
- **End-user documentation**. This type gives end users basic instructions on how to use, install and troubleshoot the software. It might provide resources, such as user guides, knowledge bases, tutorials and release notes.
- **Enterprise user documentation.** Enterprise software often has documentation for IT staff who deploy the software across the enterprise. It may also provide documentation for the end users of the software.
- **Just-in-time documentation.** This provides end users with support documentation at the exact time they will need it. This allows developers to create a minimal amount of documentation at the release of a software product and add documentation as new features are added. It is based on the Agile software development These can be knowledge bases, FAQ pages and how-to documents.

Best practices for creating software documentation

There are six common best practices for creating software documentation. They are the following:

1. **Understand user needs.** Developers must understand user needs and pain points from the start of the development process. The documentation should address those needs and provide help around pain points.
2. **Write easily understood documentation.** Documentation should be concise, simple and avoid complex jargon. It should use terms and phrases that the intended audience would use.
3. **Include internal subject matter experts.** It can help to have experienced team members and subject matter experts in the software documentation process to ensure that it is accurate.
4. **Use analytics feedback.** Analytics applications provide important feedback that can be incorporated into documentation.
5. **Ask for user feedback.** After a release, ask users what they liked and disliked about a software product and use the input to improve both the product and its documentation.

6. **Provide continuous maintenance.** As software is updated and maintained, the accompanying documentation should also be updated. Teams must constantly improve documentation as IT and user questions reveal additional needs.



Software development methodologies such as Agile follow a continuous cycle of development and improvement. Software documentation should follow a similar continuous improvement cycle as new features are rolled out.

Examples of software documentation

Some examples of software documentation include the following:

- **System documentation.** This includes architectural diagrams that detail the structure of the software and its technical design.
- **Application programming interface (API) documentation.** This is the reference documentation for calling APIs. It establishes standards for API communication and ensures that different APIs work smoothly together.
- **README files.** A README file is a high-level representation of software that usually comes with the source code.
- **Release notes.** Release notes review the new features and bug fixes included in each release of a software program.
- **How-to guides**. These take IT staff or end users through the steps needed to deploy or use the software.
- **Tutorials**. Tutorials take users through a series of steps to learn how to use the software or about a specific feature.
- **Reference documents**. These provide IT and end users with technical documentation of the software.

- **Explanations**. These clarify a particular element of the software for the user.

Software documentation tools

Various tools help vendors and developers automate the documentation process. Some important features of leading software documentation tools include the following:

- **Markdown and HTML support.** Markdown and HTML are two programming languages that software documentation is commonly written in. Markdown is an abbreviated form of HTML.
- **Feedback.** A good documentation tool will have the option to collect and review user feedback. In some cases, users contribute entire code examples. This feature may connect users and developers via email or a comments option. Some tools allow users to look at and make changes to certain code.
- **Access control.** This feature enables multiple documentation writers to contribute to one piece of documentation. It controls access with roles and permissions.
- **Click-button APIs.** With this capability, users are able to run APIs from the documentation.
- **Table of contents.** Documentation tools should enable writers to create a table of contents to simplify navigation.
- **Publishing control.** Writers can publish and unpublish pages as needed.

Some examples of documentation tools include the following:

- Apiary
- API Matic
- GitHub Pages
- ReadMe
- Stoplight
- Swagger

Software documentation must keep up with software updates in a given workflow structure. Learn best practices for creating a software update workflow for IoT devices.

Testing

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

Unit Testing

Unit testing involves the testing of each unit or an individual component of the software application. It is the first level of functional testing. The aim behind unit testing is to validate unit components with its performance.

A unit is a single testable part of a software system and tested during the development phase of the application software.

The purpose of unit testing is to test the correctness of isolated code. A unit component is an individual function or code of the application. White box testing approach used for unit testing and usually done by the developers.

Whenever the application is ready and given to the Test engineer, he/she will start checking every component of the module or module of the application independently or one by one, and this process is known as **Unit testing** or **components testing.**

Black Box Testing

Black-box testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings. This method of test can be applied virtually to every level of software testing: unit, integration, system and acceptance.

Equivalence class partitioning and boundary value analysis

Boundary value analysis is testing at the boundaries between partitions. Equivalent Class Partitioning allows you to divide set of test condition into a partition which should be considered the same**.**

White Box Testing

White-box testing is a method of software testing that tests internal structures or workings of an application, as opposed to its functionality. In white-box testing, an internal perspective of the system is used to design test cases.

Different White Box methodologies

Different types of white-boxes testing Unit Testing are Static

Analysis.

Dynamic Analysis.

Statement

Coverage. Branch

testing Coverage.

Security Testing.

Mutation Testing.

Mutation testing

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

Debugging approaches

Debugging, in computer programming and engineering, is a multistep process that involves identifying a problem, isolating the source of the problem and then either correcting the problem or determining a way to work around it. The final step of debugging is to test the correction or workaround and make sure it works.

Debugging guidelines

In the development process of any software, the software program is religiously tested, troubleshot, and maintained for the sake of delivering bug-free products. There is nothing that is error-free in the first go.

So, it's an obvious thing to which everyone will relate that as when the software is created, it contains a lot of errors; the reason being nobody is perfect and getting error in the code is not an issue, but avoiding it or not preventing it, is an issue!

All those errors and bugs are discarded regularly, so we can conclude that debugging is nothing but a process of eradicating or fixing the errors contained in a software program.

Debugging works stepwise, starting from identifying the errors, analyzing followed by removing the errors. Whenever a software fails to deliver the result, we need the software tester to test the application and solve it.

Since the errors are resolved at each step of debugging in the software testing, so we can conclude that it is a tiresome and complex task regardless of how efficient the result was.

Why do we need Debugging?

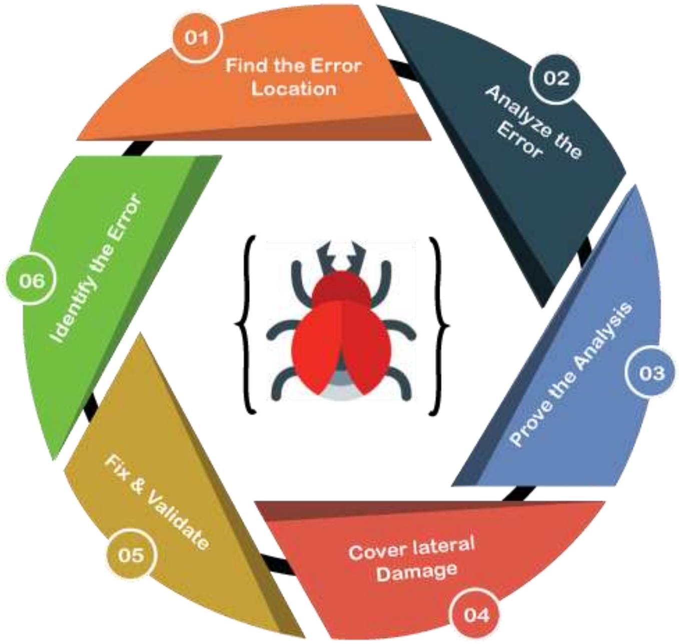
Debugging gets started when we start writing the code for the software program. It progressively starts continuing in the consecutive stages to deliver a software product because the code gets merged with several other programming units to form a software product.

Following are the benefits of Debugging:

- Debugging can immediately report an error condition whenever it occurs. It prevents hampering the result by detecting the bugs in the earlier stage, making software development stress-free and smooth.
- It offers relevant information related to the data structures that further helps in easier interpretation.
- Debugging assist the developer in reducing impractical and disrupting information.
- With debugging, the developer can easily avoid complex one-use testing code to save time and energy in software development.

Steps involved in Debugging

Following are the different steps that are involved in debugging:



1. **Identify the Error:** Identifying an error in a wrong may result in the wastage of time. It is very obvious that the production errors reported by users are hard to interpret, and sometimes the information we receive is misleading. Thus, it is mandatory to identify the actual error.
2. **Find the Error Location:** Once the error is correctly discovered, you will be required to thoroughly review the code repeatedly to locate the position of the error. In general, this step focuses on finding the error rather than perceiving it.
3. **Analyze the Error:** The third step comprises error analysis, a bottom-up approach that starts from the location of the error followed by analyzing the code. This step makes it easier to comprehend the errors. Mainly error analysis has two significant goals, i.e., evaluation of errors all over again to find existing bugs and postulating the uncertainty of incoming collateral damage in a fix.
4. **Prove the Analysis:** After analyzing the primary bugs, it is necessary to look for some extra errors that may show up on the application. By incorporating the test framework, the fourth step is used to write automated tests for such areas.
5. **Cover Lateral Damage:** The fifth phase is about accumulating all of the unit tests for the code that requires modification. As when you run these unit tests, they must pass.
6. **Fix & Validate:** The last stage is the fix and validation that emphasizes fixing the bugs followed by running all the test scripts to check whether they pass.

Debugging Strategies

- For a better understanding of a system, it is necessary to study the system in depth. It makes it easier for the debugger to fabricate distinct illustrations of such systems that are needed to be debugged. o The backward analysis analyzes the program from the backward location where the failure message has occurred to determine the defect region. It is necessary to learn the area of defects to understand the reason for defects.
- In the forward analysis, the program tracks the problem in the forward direction by utilizing the breakpoints or print statements incurred at different points in the program. It emphasizes those regions where the wrong outputs are obtained.
- To check and fix similar kinds of problems, it is recommended to utilize past experiences. The success rate of this approach is directly proportional to the proficiency of the debugger.

Software Reliability

Software Reliability

Software Reliability means **Operational reliability**. It is described as the ability of a system or component to perform its required functions under static conditions for a specific period.

Software reliability is also defined as the probability that a software system fulfills its assigned task in a given environment for a predefined number of input cases, assuming that the hardware and the input are free of error.

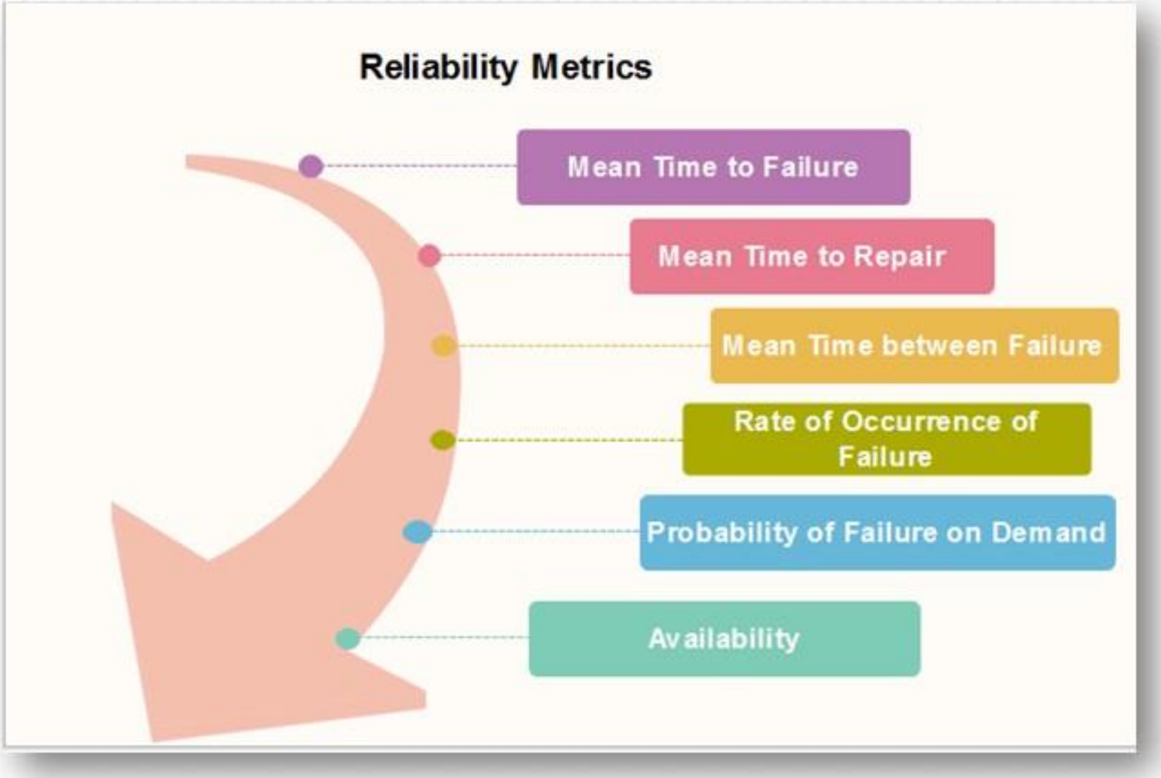
Software Reliability is an essential connect of software quality, composed with functionality, usability, performance, serviceability, capability,installability, maintainability, and documentation. Software Reliability is hard to achieve because the complexity of software turn to be high. While any system with a high degree of complexity, containing software, will be hard to reach a certain level of reliability, system developers tend to push complexity into the software layer, with the speedy growth of system size and ease of doing so by upgrading the software.

For example, large next-generation aircraft will have over 1 million source lines of software on-board; next-generation air traffic control systems will contain between one and two million lines; the upcoming International Space Station will have over two million lines on-board and over 10 million lines of ground support software; several significant life-critical defense systems will have over 5 million source lines of software. While the complexity of software is inversely associated with software reliability, it is directly related to other vital factors in software quality, especially functionality, capability, etc.

Different reliability metrics

Reliability metrics are used to quantitatively expressed the reliability of the software product. The option of which metric is to be used depends upon the type of system to which it applies & the requirements of the application domain.

Some reliability metrics which can be used to quantify the reliability of the software product are as follows:


Reliability Metrics
Mean Time to Failure
Mean Time to Repair
Mean Time between Failure
Rate of Occurrence of Failure
Probability of Failure on Demand
Availability

1. Mean Time to Failure (MTTF)

MTTF is described as the time interval between the two successive failures. An **MTTF** of 200 mean that one failure can be expected each 200-time units. The time units are entirely dependent on the system & it can even be stated in the number of transactions. **MTTF** is consistent for systems with large transactions.

For example, It is suitable for computer-aided design systems where a designer will work on a design for several hours as well as for Word-processor systems.

To measure **MTTF**, we can evidence the failure data for n failures. Let the failures appear at the time instants **t_1,t_2.....t_n.**

MTTF can be calculated as

$$\sum_{i=1}^{n} \frac{t_{i+1}-t_i}{(n-1)}$$

2. Mean Time to Repair (MTTR)

Once failure occurs, some-time is required to fix the error. **MTTR** measures the average time it takes to track the errors causing the failure and to fix them.

3. Mean Time Between Failure (MTBR)

We can merge **MTTF** & **MTTR** metrics to get the MTBF metric.

MTBF = MTTF + MTTR

Thus, an **MTBF** of 300 denoted that once the failure appears, the next failure is expected to appear only after 300 hours. In this method, the time measurements are real-time & not the execution time as in **MTTF.**

4. Rate of occurrence of failure (ROCOF)

It is the number of failures appearing in a unit time interval. The number of unexpected events over a specific time of operation. **ROCOF** is the frequency of occurrence with which unexpected role is likely to appear. A **ROCOF** of 0.02 mean that two failures are likely to occur in each 100 operational time unit steps. It is also called the failure intensity metric.

5. Probability of Failure on Demand (POFOD)

POFOD is described as the probability that the system will fail when a service is requested. It is the number of system deficiency given several systems inputs.

POFOD is the possibility that the system will fail when a service request is made.

A **POFOD** of 0.1 means that one out of ten service requests may fail.**POFOD** is an essential measure for safety-critical systems. POFOD is relevant for protection systems where services are demanded occasionally.

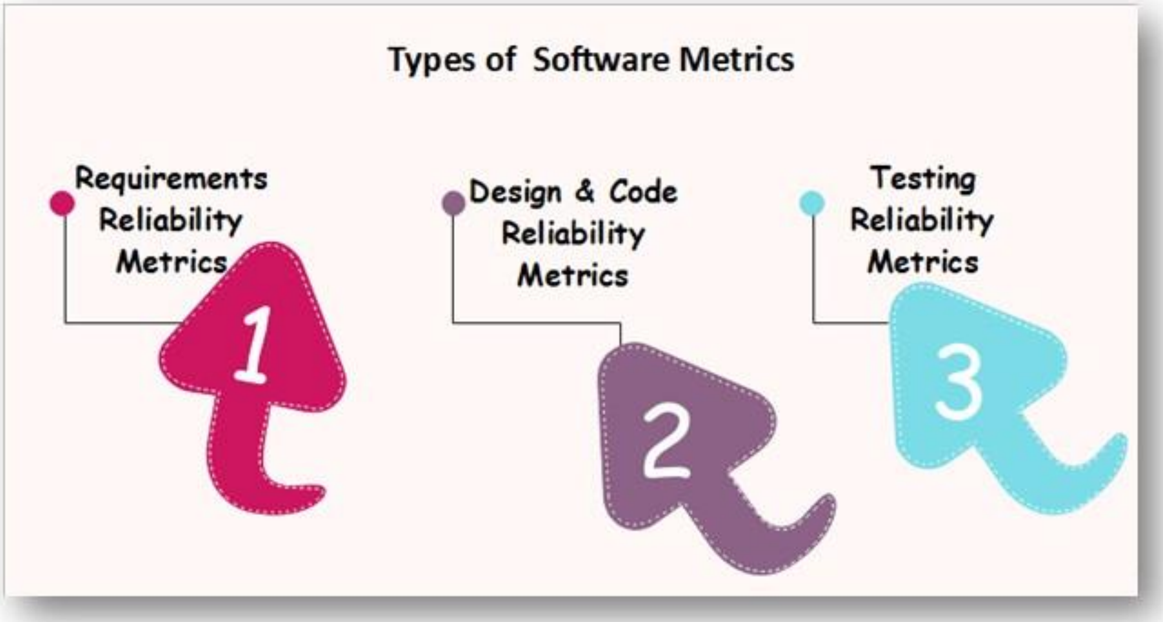
6. Availability (AVAIL)

Availability is the probability that the system is applicable for use at a given time. It takes into account the repair time & the restart time for the system. An availability of 0.995 means that in every 1000 time units, the system is feasible to be available for **995** of these. The percentage of time that a system is applicable for use, taking into account planned and unplanned downtime. If a system is down an average of four hours out of 100 hours of operation, its **AVAIL** is 96%.

Software Metrics for Reliability

The Metrics are used to improve the reliability of the system by identifying the areas of requirements.

Different Types of Software Metrics are:-


Types of Software Metrics
Requirements Reliability Metrics
1
Design & Code Reliability Metrics
2
Testing Reliability Metrics
3

Requirements Reliability Metrics

Requirements denote what features the software must include. It specifies the functionality that must be contained in the software. The requirements must be written such that is no misconception between the developer & the client. The requirements must include valid structure to avoid the loss of valuable data.

The requirements should be thorough and in a detailed manner so that it is simple for the design stage. The requirements should not include inadequate data. Requirement Reliability metrics calculates the above-said quality factors of the required document.

Design and Code Reliability Metrics

The quality methods that exists in design and coding plan are complexity, size, and modularity. Complex modules are tough to understand & there is a high probability of occurring bugs. The reliability will reduce if modules have a combination of high complexity and large size or high complexity and small size. These metrics are also available to object-oriented code, but in this, additional metrics are required to evaluate the quality.

Testing Reliability Metrics

These metrics use two methods to calculate reliability.

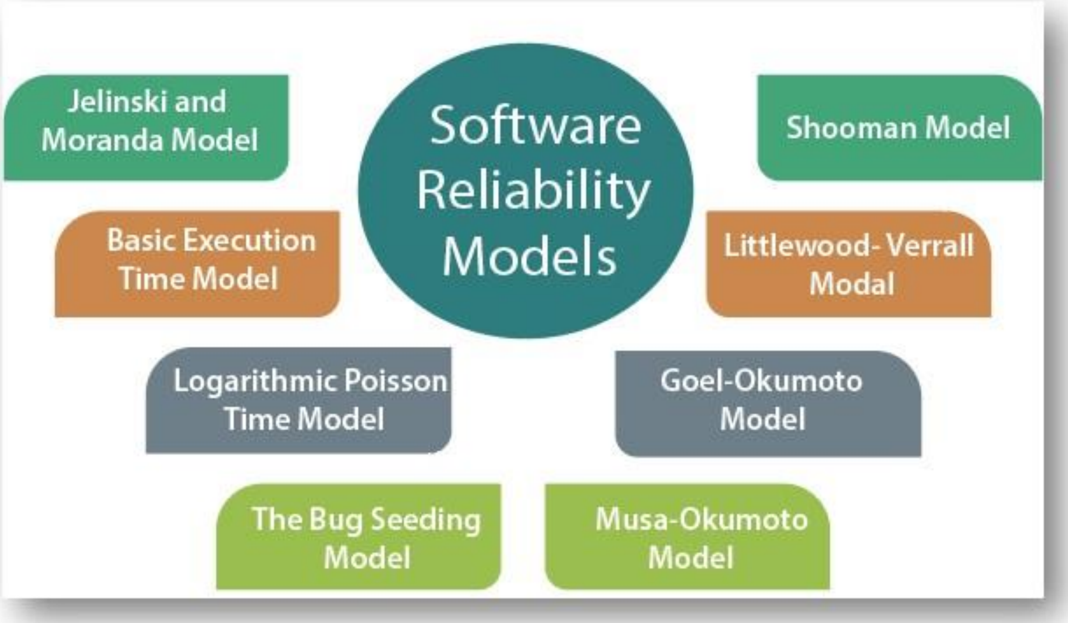
First, it provides that the system is equipped with the tasks that are specified in the requirements. Because of this, the bugs due to the lack of functionality reduces.

The **second** method is calculating the code, finding the bugs & fixing them. To ensure that the system includes the functionality specified, test plans are written that include multiple test cases. Each test method is based on one system state and tests some tasks that are based on an associated set of requirements. The goals of an effective verification program is to ensure that each elements is tested, the implication being that if the system passes the test, the requirement's functionality is contained in the delivered system.

Reliability growth modeling

A reliability growth model is a numerical model of software reliability, which predicts how software reliability should improve over time as errors are discovered and repaired. These models help the manager in deciding how much efforts should be devoted to testing. The objective of the project manager is to test and debug the system until the required level of reliability is reached.

Following are the Software Reliability Models are:


Software Reliability Models
Jelinski and Moranda Model
Shooman Model
Basic Execution Time Model
Littlewood- Verrall Modal
Logarithmic Poisson Time Model
Goel-Okumoto Model
The Bug Seeding Model
Musa-Okumoto Model

Software quality

Software quality product is defined in term of its fitness of purpose. That is, a quality product does precisely what the users want it to do. For software products, the fitness of use is generally explained in terms of satisfaction of the requirements laid down in the SRS document. Although "fitness of purpose" is a satisfactory interpretation of quality for many devices such as a car, a table fan, a grinding machine, etc.for software products, "fitness of purpose" is not a wholly satisfactory definition of quality.

Example: Consider a functionally correct software product. That is, it performs all tasks as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally right, we cannot consider it to be a quality product.

The modern view of a quality associated with a software product several quality methods such as the following:

Portability: A software device is said to be portable, if it can be freely made to work in various operating system environments, in multiple machines, with other software products, etc.

Usability: A software product has better usability if various categories of users can easily invoke the functions of the product.

Reusability: A software product has excellent reusability if different modules of the product can quickly be reused to develop new products.

Correctness: A software product is correct if various requirements as specified in the SRS document have been correctly implemented.

Maintainability: A software product is maintainable if bugs can be easily corrected as and when they show up, new tasks can be easily added to the product, and the functionalities of the product can be easily modified, etc.

Software Quality Management System

A quality management system is the principal methods used by organizations to provide that the products they develop have the desired quality.

A quality system subsists of the following:

Managerial Structure and Individual Responsibilities: A quality system is the responsibility of the organization as a whole. However, every organization has a sever quality department to perform various quality system activities. The quality system of an arrangement should have the support of the top management. Without help for the quality system at a high level in a company, some members of staff will take the quality system seriously.

Quality System Activities: The quality system activities encompass the following:

Auditing of projects

Review of the quality system

Development of standards, methods, and guidelines, etc.

Production of documents for the top management summarizing the effectiveness of the quality system in the organization.

Evolution of Quality Management System

Quality systems have increasingly evolved over the last five decades. Before World War II, the usual function to produce quality products was to inspect the finished products to remove defective devices. Since that time, quality systems of organizations have undergone through four steps of evolution, as shown in the fig. The first product inspection task gave method to quality control (QC).

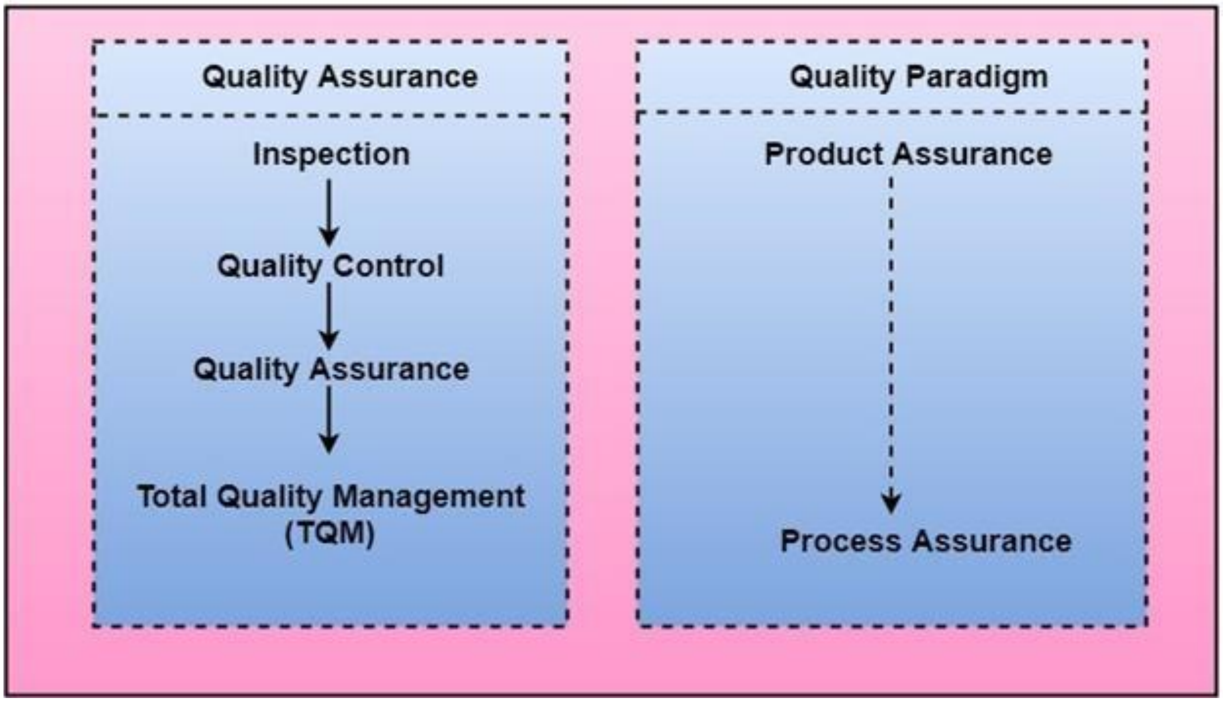
Quality control target not only on detecting the defective devices and removes them but also on determining the causes behind the defects. Thus, quality control aims at correcting the reasons for bugs and not just rejecting the products. The next breakthrough in quality methods was the development of quality assurance methods.

The primary premise of modern quality assurance is that if an organization's processes are proper and are followed rigorously, then the products are obligated to be of good quality. The new quality functions include guidance for recognizing, defining, analyzing, and improving the production process.

Total quality management (TQM) advocates that the procedure followed by an organization must be continuously improved through process measurements. TQM goes stages further than quality assurance and aims at frequently process improvement. TQM goes beyond documenting steps to optimizing them through a redesign. A term linked to TQM is Business Process Reengineering (BPR).

BPR aims at reengineering the method business is carried out in an organization. From the above conversation, it can be stated that over the years, the quality paradigm has changed from product assurance to process assurance, as shown in fig.

Evolution of quality system and corresponding shift in the quality paradigm



Software Quality Management System

Software Quality Management ensures that the required level of quality is achieved by submitting improvements to the product development process. SQA aims to develop a culture within the team and it is seen as everyone's responsibility.

Software Quality management should be independent of project management to ensure independence of cost and schedule adherences. It directly affects the process quality and indirectly affects the product quality.