Chapter - 1

INTRODUCTION TO SOFTWARE ENGINEERING

Introduction

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

Software Engineering is a systematic 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. **Correctness –**
   A software product is correct, if the different requirements as specified in the SRS document have been correctly implemented.
3. **Reusability –**
   A software product has good reusability, if the different modules of the product can easily be reused to develop new products.
4. **Testability –**
   Here software facilitates both the establishment of test criteria and the evaluation of the software with respect to those criteria.
5. **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.
6. **Portability –**
   In this case, software can be transferred from one computer system or environment to another.
7. **Adaptability –**
   In this case, software allows differing system constraints and user needs to be satisfied by making changes to the software.

Concept of System

General System Theory: It says that we have to consider each and every part of system rather than just focus on only one component of the system. Many times, the analyst tends

to focus on one component only and overlook the others, the **General System Theory tends on developing a systematic and complete structure which is further helpful in making decisions,**

SYSTEM DEFINITION: A System can be said as a collection of components that works together to achieve some common goal.

Input -> Processing -> Output

Types of System:

Open and Closed Systems

An **open system** is one that interacts with its environment and thus exchanges information, material, or energy with the environment, including random and undefined inputs. Open systems are adaptive in nature as they tend to react with the environment in such a way organizing', in the sense that they change their continued existence.

Such systems are 'self-organizing', because they change their organization in response to changing conditions. A **closed system** is one, which doesn't interact with its environment. Such systems, in business world, are rare. Thus the systems that are relatively isolated from the environment but not completely closed are termed **closed systems**.

Static and Dynamic Systems

Dynamic Systems

A **dynamic system** is composed of many interdependent subsystems. If a subsystem fails, this fact can affect the dynamics of a whole galaxy of other subsystems and the survival of the main system.

A dynamic system can become static or unstable and destructive. Every system has a finite amount of energy with which to survive, therefore instability cannot last indefinitely.

Static Systems

These are the models that gives us a picture about some relationship.

Programs v/s Software Products

Program

1. Program is a set of instructions written in a programming language used to execute for a specific task or particular function.
2. A program does not have further categorization.
3. A program cannot be software.
4. A program consists of a set of instructions which are coded in a programming language like c, C++, PHP, Java etc.
5. Programs do not have a user interface.
6. A program is developed and used by either a single programmer or a group of programmers.
7. A program is compiled every time when we need to generate some output from it.
8. Program has limited functionality and less features.
9. Program functionality is dependent on compiler.
10. A program takes less time to build/make.
11. Program development approach is un-procedural, un-organized and unplanned.
12. The size of a program ranges from kilobytes (Kb) to megabytes (Mb).
13. Examples of computer programs include: Operating system, office suite, video games, malware, a web browser like Mozilla Firefox and Apple Safari.

Software Products

1. Software is a collection of several programs and other procedures and documentation.
2. Software can be categorized into two categories: application software and system software.
3. Software can be a program.
4. Software consists of bundles of programs and data files. Programs in specific software use these data files to perform a dedicated type of tasks.
5. Every software has a dedicated user interface. The user interface of software may be in the form of a command prompt or in a graphical format.
6. Software is developed by either a single programmer or a group of programmers.
7. Whole software is compiled, tested and debugged during the development process.
8. Software has lots of functionality and features such as GUI, input/output data, process etc.
9. Software functionality is dependent on the operating system.
10. Software takes relatively more time to build/make when compared to program.
11. Software development approach is systematic, organized and very well planned.
12. The size of a software ranges from megabytes (Mb) to Gigabytes (Gb).
13. Examples of software include: Microsoft Word, Microsoft Excel, VLC media player, Firefox, Adobe Reader, Windows, Linux, Unix, Mac etc.

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

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, **flowchartingtechnique** 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.

Note that having more GOTO constructs in the flowchart makes the control flow messy, which makes it difficult to understand and debug. In order to provide clarity of control flow, the use of GOTO constructs in flowcharts should be avoided and **structured constructs-decision,** sequence, and loop-should be used to develop **structured flowcharts.** The decision structures are used for conditional execution of statements (for example, if statement). The sequence structures are used for the sequentially executed statements. The loop structures

are used for performing some repetitive tasks in the program. The use of structured constructs formed the basis of the **structured programming** methodology.

Structured programming became a powerful tool that allowed programmers to write moderately complex programs easily. It forces a logical structure in the program to be written in an efficient and understandable manner. The purpose of structured programming is to make the software code easy to modify when required. Some languages such as Ada, Pascal, and dBase are designed with features that implement the logical program structure in the software code.

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-flow-oriented** technique came into existence. In this technique, the flow of data through business functions or processes is represented using **Data-flowDiagram (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.

Software Development Life Cycle, SDLC for short, is a well-defined, structured sequence of stages in software engineering to develop the intended software product.

SDLC Activities

SDLC provides a series of steps to be followed to design and develop a software product efficiently. SDLC framework includes the following steps:

Communication

Requirement Gathering

Feasibility Study

System Analysis

Software Design

SDLC

Coding

Testing

Integration

Implementation

Operations & Maintenance

Disposition

Communication

This is the first step where the user initiates the request for a desired software product. He contacts the service provider and tries to negotiate the terms. He submits his request to the service providing organization in writing.

Requirement Gathering

This step onwards the software development team works to carry on the project. The team holds discussions with various stakeholders from problem domain and tries to bring out as much information as possible on their requirements. The requirements are contemplated and segregated into user requirements, system requirements and functional requirements. The requirements are collected using a number of practices as given -

- studying the existing or obsolete system and software,

- conducting interviews of users and developers,

- referring to the database or

- Collecting answers from the questionnaires.

Feasibility Study

After requirement gathering, the team comes up with a rough plan of software process. At this step the team analyzes if a software can be made to fulfill all requirements of the user and if there is any possibility of software being no more useful. It is found out, if the project is financially, practically and technologically feasible for the organization to take up. There are many algorithms available, which help the developers to conclude the feasibility of a software project.

System Analysis

At this step the developers decide a roadmap of their plan and try to bring up the best software model suitable for the project. System analysis includes Understanding of software product limitations, learning system related problems or changes to be done in existing systems beforehand, identifying and addressing the impact of project on organization and personnel etc. The project team analyzes the scope of the project and plans the schedule and resources accordingly.

Software Design

Next step is to bring down whole knowledge of requirements and analysis on the desk and design the software product. The inputs from users and information gathered in requirement gathering phase are the inputs of this step. The output of this step comes in the form of two designs; logical design and physical design. Engineers produce meta-data and data dictionaries, logical diagrams, data-flow diagrams and in some cases pseudo codes.

Coding

This step is also known as programming phase. The implementation of software design starts in terms of writing program code in the suitable programming language and developing error-free executable programs efficiently.

Testing

An estimate says that 50% of whole software development process should be tested. Errors may ruin the software from critical level to its own removal. Software testing is done while coding by the developers and thorough testing is conducted by testing experts at various levels of code such as module testing, program testing, product testing, in-house testing and testing the product at user's end. Early discovery of errors and their remedy is the key to reliable software.

Integration

Software may need to be integrated with the libraries, databases and other program(s). This stage of SDLC is involved in the integration of software with outer world entities.

Implementation

This means installing the software on user machines. At times, software needs post-installation configurations at user end. Software is tested for portability and adaptability and integration related issues are solved during implementation.

Operation and Maintenance

This phase confirms the software operation in terms of more efficiency and less errors. If required, the users are trained on, or aided with the documentation on how to operate the software and how to keep the software operational. The software is maintained timely by updating the code according to the changes taking place in user end environment or technology. This phase may face challenges from hidden bugs and real-world unidentified problems.

Disposition

As time elapses, the software may decline on the performance front. It may go completely obsolete or may need intense up gradation. Hence a pressing need to eliminate a major portion of the system arises. This phase includes archiving data and required software

components, closing down the system, planning disposition activity and terminating system at appropriate end-of-system time.


Classical waterfall model

The classical waterfall model is one of the oldest software lifecycle models that were developed to provide a systematic approach to the developers for developing any software. It can be said that all the other software lifecycle models were derived from this classical waterfall model just by adding some additional features and omitting some.

The name **waterfall** itself defines the characteristics of the model. As a waterfall flows only downwards and cannot flow back, in the same way, the phases in the **waterfall model** are followed one after the other in the same sequence as mentioned. And after completing a phase and entering into another phase, we cannot go back to the previous phase.

The **different phases that are included in the classical waterfall model** are:

- Feasibility study
- Requirement analysis and specification
- Design
- Coding and unit testing
- Integration and system testing
- Maintenance

An overview of these **phases of the classical waterfall model** can be made from the following diagram:
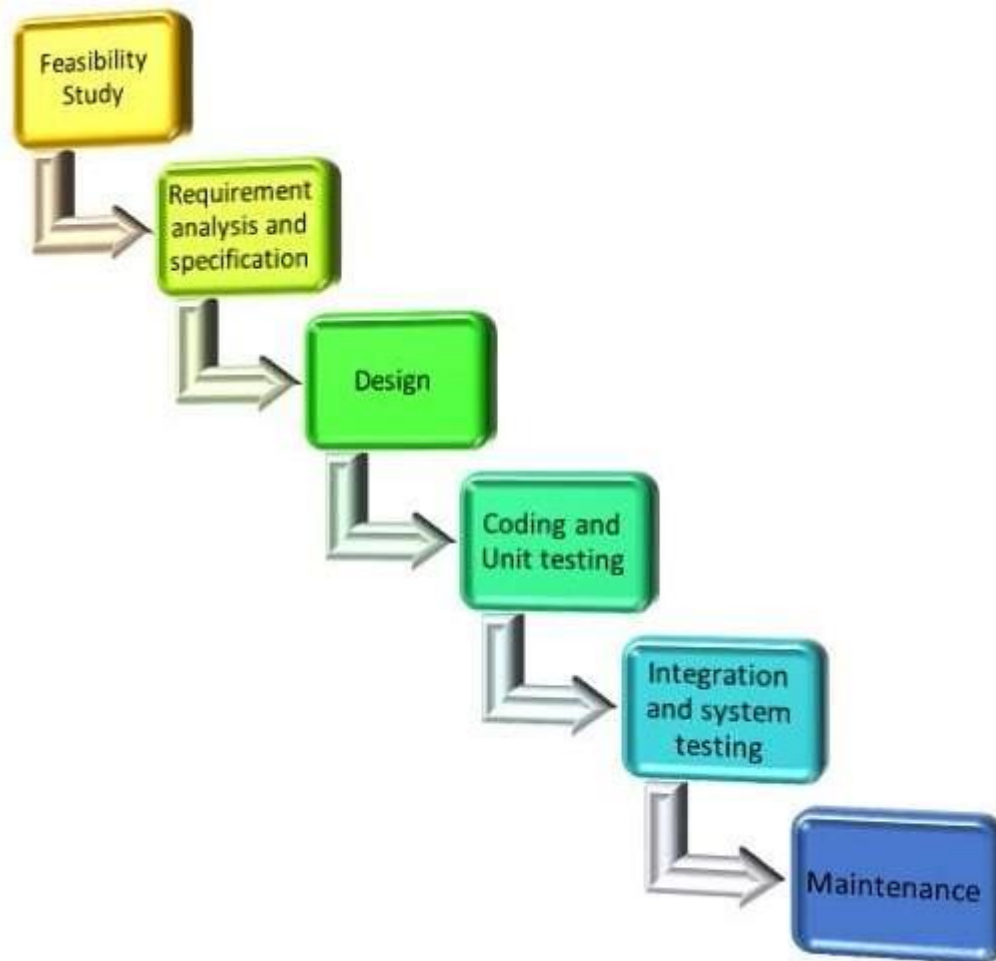
Each of the phases of this model is executed one after the other. You cannot break the flow and you cannot go back to the previous phase as stated earlier.

This means that if you are in the design phase, then you cannot conduct the requirement analysis and specification study for the software. That has already been earlier and now there is no scope left for making any kind of changes in the software requirements according to **the classical waterfall model**. All that needs to be completed before getting onto the design phase.

We can also relate this model with our many days to day activities that must be performed in a particular order. This can be well understood with the following example:

Suppose you want to make a dish in a couple of hours. Then your feasibility phase and requirement analysis phase starts in the initial phase of making the dish itself. First, you need to decide what you will be making. That is your feasibility study. Now let's say your requirements include chopping board, vegetables, flour, spices, oil, etc. You need to list them up before going to the market itself. That's your requirements and specification phase. Now you prepare everything up and assemble in the utensil. That's the designing part. And the cooking and tasting part becomes your coding and testing part. Final plating of the dish in the integration part and then you finally present your dish in front of the customers. After that, adding any sort of extra spices or re-heating the dish in some cases can be thought as of your maintenance part.

So you see, if you skip any of the phase-in between or break the flow of these phases, then you will not be able to present your final dish in front of the customers on time.

Prototype Model

The Prototype model is one of the software development life cycle models in which a prototype is built with minimal requirements, which is then tested and modified based on the feedback received from the client until a final prototype with desired functionalities gets created. This final prototype also acts as a base for the final product.

As mentioned earlier, this model is useful when all the detailed requirements are not known to the client before starting the project. It is also useful when the product to be developed is a complex one and similar product does not exist in the market.

In such a scenario, the client can ask the developers to start working on the basic prototype with limited requirements. Once the basic prototype is ready, the client can see and check the prototype to decide what all changes are required.

The client can also use the prototype to do market research and gather end-user or customer feedback.

When the client has decided about the changes that need to be made, the client will give these requirements to the requirements gathering team, which eventually reach the development

team.

Developers can then start working on the modifications to the basic prototype. This cycle will be repeated until the client is satisfied with the prototype which reflects the final product.

Phases of Prototype Model

The following are the primary phases involved in the development cycle of any prototype model.

- **Initial Communication** – In this phase, business analysts and other individuals responsible for collecting the requirements and discussing the need for the product, meet the stakeholders or clients.
- **Quick Plan** – Once basic requirements have been discussed, a quick plan of the initial prototype is made.
- **Modeling Quick Design** – User interface part i.e. designing part of the prototype is carried out in this phase.
- **Development of the Prototype** – In this phase, the designed prototype is coded and developed.
- **Deployment, Delivery, and Feedback of the Prototype** – In this phase, the initial prototype is deployed and is accessible to clients for its use. Clients review or evaluate the prototype and they provide their feedback to the requirements gathering and development teams.

  Above mentioned phases keep repeating until the replica of the final product is deployed.
- **Final Product Design, Implementation, Testing, Deployment, and Maintenance** – Once the client finalizes a prototype, on the basis of the prototype, the final product is designed and developed. This developed product is tested by the testing team and if it is ready to go LIVE, the product is deployed and is available for end-user.

Types of Prototype Model

Developers can choose from available prototype model types based on the product's requirements that have been covered in this section, let's look at them.

- **Rapid Throwaway Prototyping** – In this method, the prototype is developed rapidly based on the initial requirements and given to the client for review. Once the client

provides feedback, final requirements are updated and work on the final product begins. As the name suggests, the developed prototype is discarded, and it will not be part of the final product. It is also known as close-ended prototyping.

- **Evolutionary Prototyping** – In this method, a prototype is made, and the client feedback is received. Based on the feedback, the prototype is refined until the client considers it the final product. It follows an incremental development approach and saves time compared to the rapid throwaway prototyping method as in evolutionary prototyping old prototype is reworked rather than developing a new prototype from scratch. It is also known as breadboard prototyping.

- **Incremental Prototyping** – In this type of prototype model, final product requirements are break into smaller parts and each part is developed as a separate prototype. In the end, all the parts (prototypes) are merged which becomes a final product.

- **Extreme Prototyping** – This type of prototyping model is mainly used for web applications. It is divided into three phases-

  - First basic prototype with static pages is created, it consists of HTML pages.
  - Next, using a services layer, data processing is simulated.
  - In the last phase, services are implemented.

Advantages of Prototype Model

Prototype model offers the following benefits-

- Quick client feedback is received which speeds up the development process. Also, it helps the development team to understand the client's needs.
- Developed prototypes can be used later for any similar projects.
- Any missing functionality and any error can be detected early.
- It is useful when requirements are not clear from the client's end, even with limited requirements, the development team can start the development process.

**Disadvantages of Prototype Model**

Apart from appealing advantages, the prototype model has many disadvantages that are listed below-

- It is a time-consuming process or method as multiple prototypes might be needed until the client reaches the final requirements. The Client may not have an explicit idea about what they want.

- This method involves too much client interaction and involvement, which can be done only with a committed client.

- In the beginning, it is a bit difficult to predict the exact amount of time needed to reach the final product.

- While coding, developers do not have a broad perspective of what is coming, because of which they might use an underlying architecture that is not suitable for a final product.

- To produce the quick prototype, developers might make weak decisions during the development process (especially implementation decisions), and compromise on quality which might eventually affect the product.

Spiral Model

Spiral Model is a combination of a waterfall model and iterative model. Each phase in spiral model begins with a design goal and ends with the client reviewing the progress. The spiral model was first mentioned by Barry Boehm in his 1986 paper.

The development team in Spiral-SDLC model starts with a small set of requirement and goes through each development phase for those set of requirements. The software engineering team adds functionality for the additional requirement in every-increasing spirals until the application is ready for the production phase.
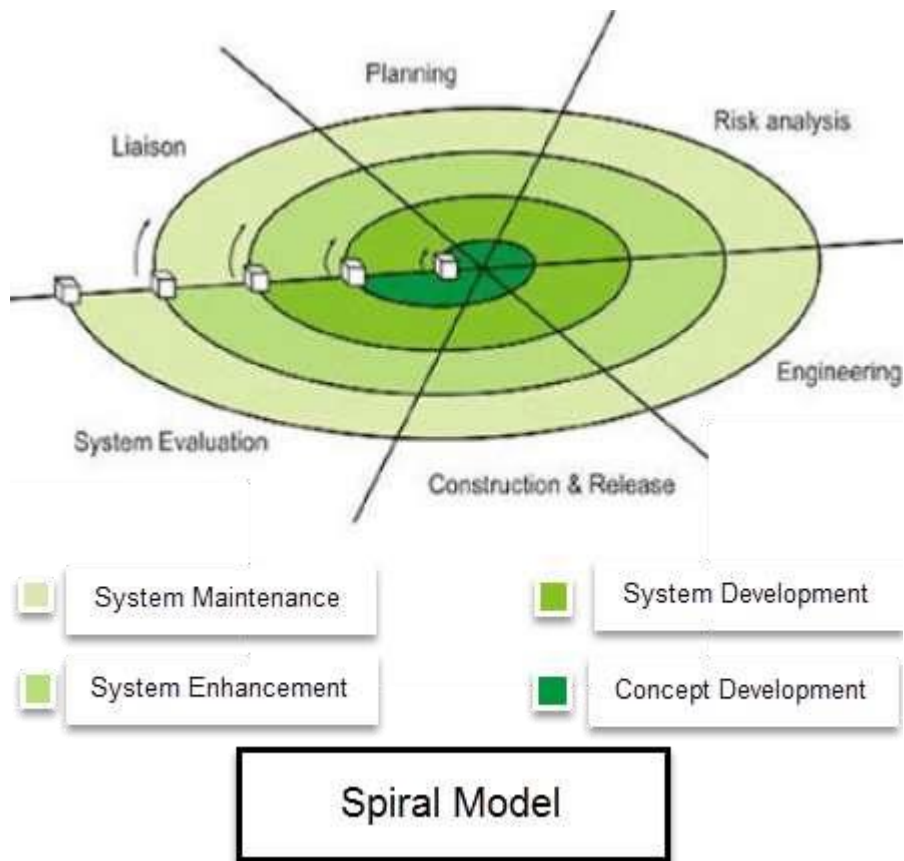
Spiral Model Phases

Spiral Model Phases	Activities performed during phase
Planning	• It includes estimating the cost, schedule and resources for the iteration. It also involves understanding the system requirements for continuous communication between the system analyst and the customer
Risk Analysis	• Identification of potential risk is done while risk mitigation strategy is planned and finalized
Engineering	• It includes testing, coding and deploying software at the customer site

Evaluation	• Evaluation of software by the customer. Also, includes identifying and monitoring risks such as schedule slippage and cost overrun

When to use Spiral Methodology?

- When project is large
- When releases are required to be frequent
- When creation of a prototype is applicable
- When risk and costs evaluation is important
- For medium to high-risk projects
- When requirements are unclear and complex
- When changes may require at any time
- When long term project commitment is not feasible due to changes in economic priorities

Advantages and Disadvantages of Spiral Model

Advantages	Disadvantages
• Additional functionality or changes can be done at a later stage	• Risk of not meeting the schedule or budget
• Cost estimation becomes easy as the prototype building is done in small fragments	• It works best for large projects only also demands risk assessment expertise
• Continuous or repeated development helps in risk management	• For its smooth operation spiral model protocol needs to be followed strictly
• Development is fast and features are added in a systematic way	• Documentation is more as it has intermediate phases

- There is always a space for customer feedback
- It is not advisable for smaller project, it might cost them a lot

Difference between Waterfall Model and Spiral Model

While in the spiral model, the customer is made aware of all the happenings in the software development, in the waterfall model the customer is not involved. This often leads to situations, where the software is not developed according to the needs of the customer. In the spiral model, the customer is involved in the software development process from the word go. This helps in ensuring that the software meets the needs of the customer.

In the waterfall model, when the development process shifts to the next stage, there is no going back. This often leads to roadblocks, especially during the coding phase. Many times it is seen that the design of the software looks feasible on paper, however, in the implementation phase it may be difficult to code for the same. However, in the spiral model, since there are different iterations, it is rather easier to change the design and make the software feasible.

In the spiral model, one can revisit the different phases of software development, as many times as one wants, during the entire development process. This also helps in back tracking, reversing or revising the process. However, the same is not possible in the waterfall model, which allows no such scope.

Often people have the waterfall model or spiral model confusion due to the fact, that the spiral model seems to be a complex model. It can be attributed to the fact that there are many iterations, which go into the model. At the same time, often there is no documentation involved in the spiral model, which makes it difficult to keep a track of the entire process. On the other hand, the waterfall model has sequential progression, along with clear documentation of the entire process. This ensures one has a better hold over the entire process.

From the above discussion on spiral model vs waterfall model, it is clear that both the models have their own advantages and shortcomings. While one is stuck between the waterfall model vs spiral model debate, it is best to evaluate the software that is being developed and then decide the right approach. The size of the project and the urgency of the software will have to be taken into consideration for the same. At the same time, the resources available will have an important role to play in the software development process.

Prototype Model Vs. Waterfall Model

Now that you have a basic understanding of what the waterfall model and prototype model are all about, let me point out the prime differences in these two software design philosophies. The waterfall model directly delivers the final product to the user and his feedback is only taken in, before the design phase. Conversely, the prototype model creates several rough working applications and involves constant user interaction, until the developers come up with the final application, which satisfies the user.

While the waterfall model is linear, the prototype model is non linear and evolutionary in nature. Both processes have their merits and demerits. According to experts, the prototype model is well suited for online applications where user interfaces are the most important component and clients are not clear about what they exactly need in the final product.

On the other hand, the waterfall model is better suited for more conventional software projects, where user requirements are clear, right from the start. A prototype model ensures users involvement which makes last minute changes possible. The waterfall model makes it difficult to implement any changes suggested by the user, after initial specification.

To conclude, it's apparent that prototype model is best suited when the client himself is not sure of what he wants and waterfall model is a safe bet, when the end user or client is clear about what he wants. Before deciding which model would be ideally suited for your own software development project, study the nature of the client requirements and choose a process which would give you the best chances of creating a satisfying end product.

UNIT-3

Software Planning

Before starting a software project, it is essential to determine the tasks to be performed and properly manage allocation of tasks among individuals involved in the software development. Hence, planning is important as it results in effective software development.

Project planning is an organized and integrated management process, which focuses on activities required for successful completion of the project. It prevents obstacles that arise in the project such as changes in projects or organization's objectives, non-availability of resources, and so on. Project planning also helps in better utilization of resources and optimal usage of the allotted time for a project. The other objectives of project planning are listed below.

- It defines the roles and responsibilities of the project management team members.
- It ensures that the project management team works according to the business objectives.
- It checks feasibility of the schedule and user requirements.
- It determines project constraints.

Responsibilities of Software Project Manager

Software project manager is the most important person inside a team who takes the overall responsibilities to manage the software projects and play an important role in the successful completion of the projects. A project manager has to face many difficult situations to accomplish these works. In fact, the job responsibilities of a project manager range from invisible activities like building up team morale to highly visible customer presentations. Most of the managers take responsibility for writing the project proposal, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, managerial report writing and presentation and interfacing with clients. The task of a project manager are classified into two major types:

1. Project planning
2. Project monitoring and control

Project planning

Project planning is undertaken immediately after the feasibility study phase and before the starting of the requirement analysis and specification phase. Once a project has been found to be feasible, Software project managers started project planning. Project planning is completed before any development phase starts. Project planning involves estimating several characteristics of a project and then plans the project activities based on these estimations. Project planning is done with most care and attention. A wrong estimation can result in schedule slippage. Schedule delay can cause customer dissatisfaction, which may lead to a project failure. For effective project planning, in addition to a very good knowledge of various estimation techniques, past experience is also very important. During the project planning the project manager performs the following activities:

1. **Project Estimation:** Project Size Estimation is the most important parameter based on which all other estimations like cost, duration and effort are made.

   - **Cost Estimation:** Total expenses to develop the software product is estimated.
   - **Time Estimation:** The total time required to complete the project.
   - **Effort Estimation:** The effort needed to complete the project is estimated.

   The effectiveness of all later planning activities is dependent on the accuracy of these three estimations.

2. **Scheduling:** After completion of estimation of all the project parameters, scheduling for manpower and other resources are done.

3. **Staffing:** Team structure and staffing plans are made.

4. **Risk Management:** The project manager should identify the unanticipated risks that may occur during project development risk, analysis the damage might cause these risks and take risk reduction plan to cope up with these risks.

5. **Miscellaneous plans:** This includes making several other plans such as quality assurance plan, configuration management plan, etc.

The order in which the planning activities are undertaken is shown in the below figure:
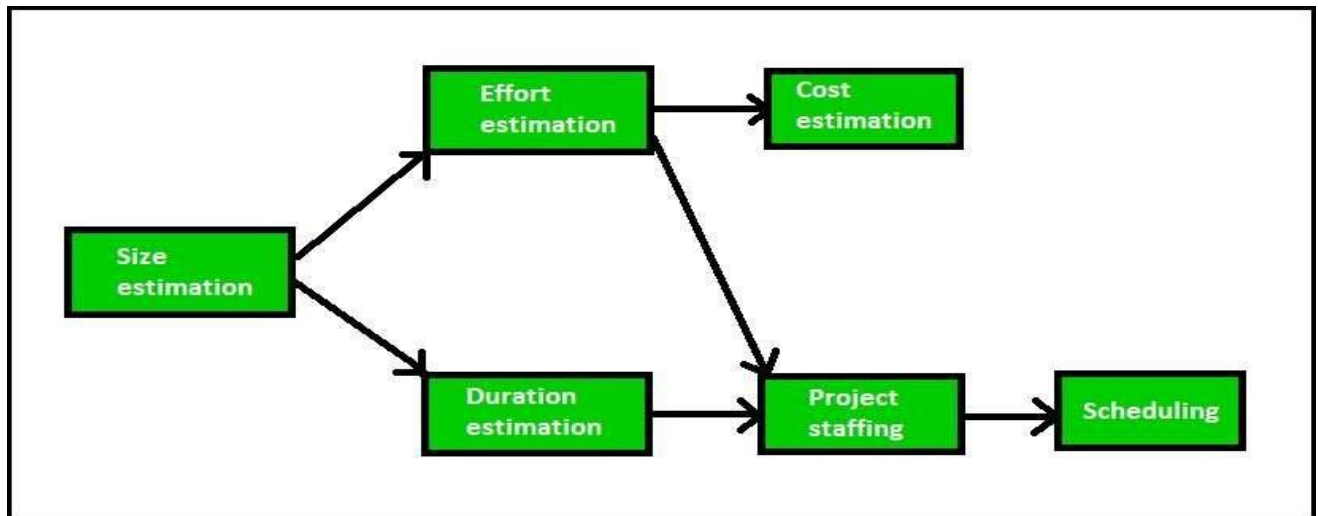
Project monitoring and control

Project monitoring and control activities are undertaken once the development activities start. The main focus of project monitoring and control activities is to ensure that the software development proceeds as per plan. This includes checking whether the project is going on as per plan or not if any problem created then the project manager must take necessary action to solve the problem.

Role of a software project manager: There are many roles of a project manager in the development of software.

- **Lead the team:** The project manager must be a good leader who makes a team of different members of various skills and can complete their individual task.
- **Motivate the team-member:** One of the key roles of a software project manager is to encourage team member to work properly for the successful completion of the project.
- **Tracking the progress:** The project manager should keep an eye on the progress of the project. A project manager must track whether the project is going as per plan or not. If any problem arises, then take necessary action to solve the problem. Moreover, check whether the product is developed by maintaining correct coding standards or not.
- **Liaison:** Project manager is the link between the development team and the customer. Project manager analysis the customer requirements and convey it to the development team and keep telling the progress of the project to the customer.

Moreover, the project manager checks whether the project is fulfilling the customer requirements or not.

- **Documenting project report:** The project manager prepares the documentation of the project for future purpose. The reports contain detailed features of the product and various techniques. These reports help to maintain and enhance the quality of the project in the future.

Necessary skills of software project manager: A good theoretical knowledge of various project management techniques is needed to become a successful project manager, but only theoretical knowledge is not enough. Moreover, a project manager must have good decision-making abilities, good communication skills and the ability to control the team members with keeping a good rapport with them and the ability to get the work done by them. Some skills such as tracking and controlling the progress of the project, customer interaction, good knowledge of estimation techniques and previous experience are needed.

Skills that are the most important to become a successful project manager are given below:

- Knowledge of project estimation techniques
- Good decision-making abilities at the right time
- Previous experience of managing a similar type of projects
- Good communication skill to meet the customer satisfaction
- A project manager must encourage all the team members to successfully develop the product
- He must know the various type of risks that may occur and the solution for these problems

Project Size Estimation

It's important to understand that *project size estimation* is the most fundamental parameter. If this is estimated accurately then all other parameters like effort, duration, cost, etc can be determined easily.

At present two *techniques* that are used to estimate project size are:

1. **Lines of code or LOC**
2. **Function point**

Both of the above serves as important *project size estimation metrics*.

Lines of code

Lines of code or LOC is the most popular and used metrics to estimate size. LOC determination is simple as well. LOC measures the project size in terms of number of lines of statements or instructions written in the source code. In this count, comments and headers are ignored.

Shortcomings of LOC

- Estimating LOC by analyzing the problem specification is difficult. Estimation of accurate LOC is only possible once the complete code has been developed. As project planning needs to be done before the development work begins so this metrics is of little use for project managers.
- Two different source files having same number of lines may not require same effort. The file having complex logic would require more effort than one with simple logic. Based on LOC proper estimation may not be possible.
- LOC is the numerical measurement of problem size. This metrics will vary to a large extent from programmer to programmer. An experienced professional may write same logic in less number of lines than a novice programmer.

Function point metrics

Function point metrics overcomes many of the shortcomings of LOC.

Function point metrics proposes that size of the software project is directly dependent on various functionalities it supports. More the features supported the more would be the size.

This technique helps determine size of the project directly from the problem specification so is really helpful to project managers during project planning while determining size.

Function Point

It is widely used to measure the size of software. Function Point concentrates on functionality provided by the system. Features and functionality of the system are used to measure the software complexity.

Function point counts on five parameters, named as External Input, External Output, Logical Internal Files, External Interface Files, and External Inquiry. To consider the complexity of software each parameter is further categorized as simple, average or complex.
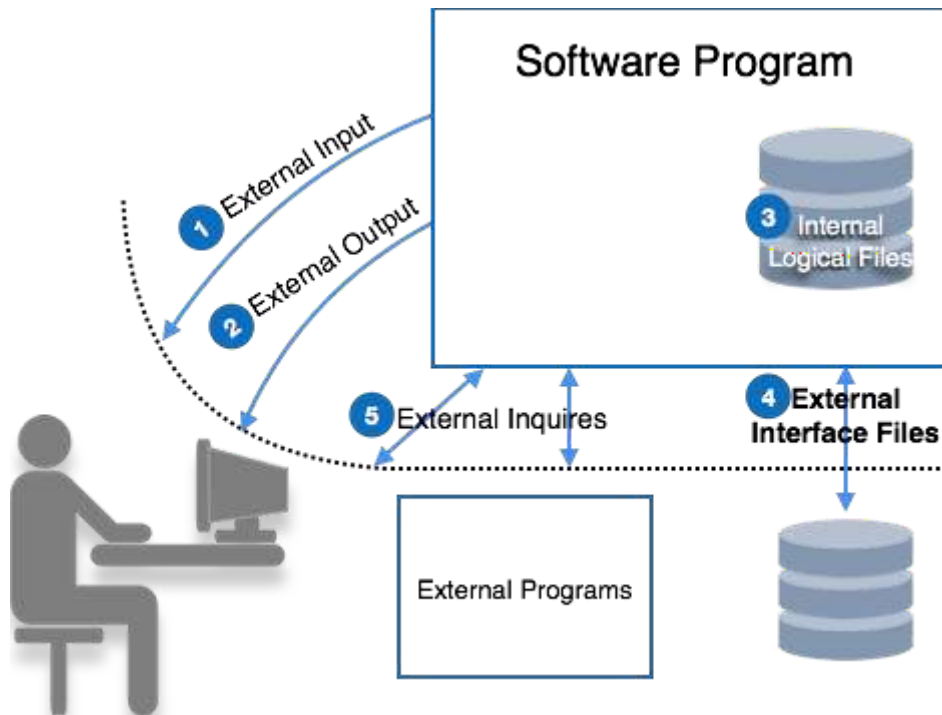
Let us see parameters of function point:

External Input

Every unique input to the system, from outside, is considered as external input. Uniqueness of input is measured, as no two inputs should have same formats. These inputs can either be data or control parameters.

- **Simple** - if input count is low and affects less internal files

- **Complex** - if input count is high and affects more internal files

- **Average** - in-between simple and complex.

External Output

All output types provided by the system are counted in this category. Output is considered unique if their output format and/or processing are unique.

- **Simple** - if output count is low

- **Complex** - if output count is high

- **Average** - in between simple and complex.

Logical Internal Files

Every software system maintains internal files in order to maintain its functional information and to function properly. These files hold logical data of the system. This logical data may contain both functional data and control data.

- **Simple** - if number of record types are low
- **Complex** - if number of record types are high
- **Average** - in between simple and complex.

External Interface Files

Software system may need to share its files with some external software or it may need to pass the file for processing or as parameter to some function. All these files are counted as external interface files.

- **Simple** - if number of record types in shared file are low
- **Complex** - if number of record types in shared file are high
- **Average** - in between simple and complex.

External Inquiry

An inquiry is a combination of input and output, where user sends some data to inquire about as input and the system responds to the user with the output of inquiry processed. The complexity of a query is more than External Input and External Output. Query is said to be unique if its input and output are unique in terms of format and data.

- **Simple** - if query needs low processing and yields small amount of output data
- **Complex** - if query needs high process and yields large amount of output data
- **Average** - in between simple and complex.

Each of these parameters in the system is given weightage according to their class and complexity. The table below mentions the weightage given to each parameter:

Parameter	Simple	Average	Complex
Inputs	3	4	6
Outputs	4	5	7
Enquiry	3	4	6
Files	7	10	15
Interfaces	5	7	10

The table above yields raw Function Points. These function points are adjusted according to the environment complexity. System is described using fourteen different characteristics:

- Data communications
- Distributed processing
- Performance objectives
- Operation configuration load
- Transaction rate
- Online data entry,
- End user efficiency
- Online update
- Complex processing logic
- Re-usability
- Installation ease
- Operational ease
- Multiple sites
- Desire to facilitate changes

These characteristics factors are then rated from 0 to 5, as mentioned below:

- No influence
- Incidental

- Moderate

- Average

- Significant

- Essential

All ratings are then summed up as N. The value of N ranges from 0 to 70 (14 types of characteristics x 5 types of ratings). It is used to calculate Complexity Adjustment Factors (CAF), using the following formulae:

CAF $= 0.65 + 0.01N$

Then,

Delivered Function Points (FP)$=$ CAF x Raw FP

This FP can then be used in various metrics, such as:

Cost $= \$ / FP$

Quality $=$ Errors $/ FP$

Productivity $=$ FP $/$ person-month

COCOMO Model

Boehm proposed COCOMO (Constructive Cost Estimation Model) in 1981.COCOMO is one of the most generally used software estimation models in the world. COCOMO predicts the efforts and schedule of a software product based on the size of the software.

The necessary steps in this model are:

1. Get an initial estimate of the development effort from evaluation of thousands of delivered lines of source code (KDLOC).

2. Determine a set of 15 multiplying factors from various attributes of the project.

3. Calculate the effort estimate by multiplying the initial estimate with all the multiplying factors i.e., multiply the values in step1 and step2.

The initial estimate (also called nominal estimate) is determined by an equation of the form used in the static single variable models, using KDLOC as the measure of the size. To determine the initial effort E_i in person-months the equation used is of the type is shown below

$E_i=a*(KDLOC)b$

The value of the constant a and b are depends on the project type.

In COCOMO, projects are categorized into three types:

1. Organic
2. Semidetached
3. Embedded

1.Organic: A development project can be treated of the organic type, if the project deals with developing a well-understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar methods of projects. **Examples of this type of projects are simple business systems, simple inventory management systems, and data processing systems.**

2. Semidetached: A development project can be treated with semidetached type if the development consists of a mixture of experienced and inexperienced staff. Team members may have finite experience in related systems but may be unfamiliar with some aspects of the order being developed. **Example of Semidetached system includes developing a new operating system (OS), a Database Management System (DBMS), and complex inventory management system.**

3. Embedded: A development project is treated to be of an embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational method exist. **For Example:** ATM, Air Traffic control.

For three product categories, Bohem provides a different set of expression to predict effort (in a unit of person month)and development time from the size of estimation in KLOC(Kilo Line of code) efforts estimation takes into account the productivity loss due to holidays, weekly off, coffee breaks, etc.

According to Boehm, software cost estimation should be done through three stages:

1. Basic Model
2. Intermediate Model
3. Detailed Model

1. Basic COCOMO Model: The basic COCOMO model provides an accurate size of the project parameters. The following expressions give the basic COCOMO estimation model:

$$\text{Effort} = a_1 * (\text{KLOC})^{a_2} \text{ PM}$$
$$\text{Tdev} = b_1 * (\text{efforts})^{b_2} \text{ Months}$$

Where

KLOC is the estimated size of the software product indicate in Kilo Lines of Code,

a_1, a_2, b_1, b_2 are constants for each group of software products,

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

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

Estimation of development effort

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

Organic: Effort = $2.4(\text{KLOC})^{1.05}$ PM

Semi-detached: Effort = $3.0(\text{KLOC})^{1.12}$ PM

Embedded: Effort = $3.6(\text{KLOC})^{1.20}$ PM
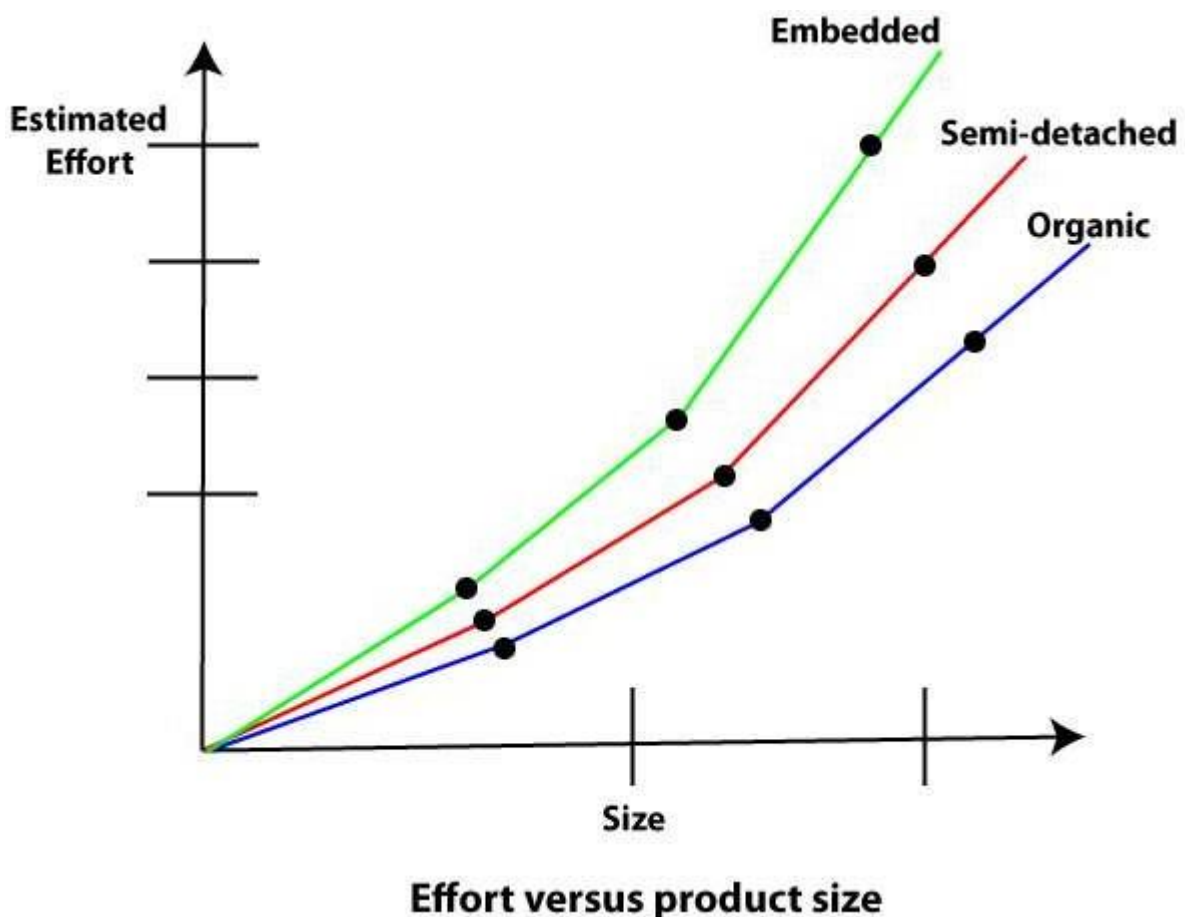
Estimation of development time

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

Organic: Tdev = 2.5(Effort) 0.38 Months

Semi-detached: Tdev = 2.5(Effort) 0.35 Months

Embedded: Tdev = 2.5(Effort) 0.32 Months

Some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes. Fig shows a plot of estimated effort versus product size. From fig, we can observe that the effort is somewhat superliner in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.

Effort versus product size

The development time versus the product size in KLOC is plotted in fig. From fig it can be observed that the development time is a sub linear function of the size of the product, i.e. when the size of the product increases by two times, the time to develop the product does not double but rises moderately. This can be explained by the fact that for larger products, a

larger number of activities which can be carried out concurrently can be identified. The parallel activities can be carried out simultaneously by the engineers. This reduces the time to complete the project. Further, from fig, it can be observed that the development time is roughly the same for all three categories of products. For example, a 60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semidetached, or embedded type.
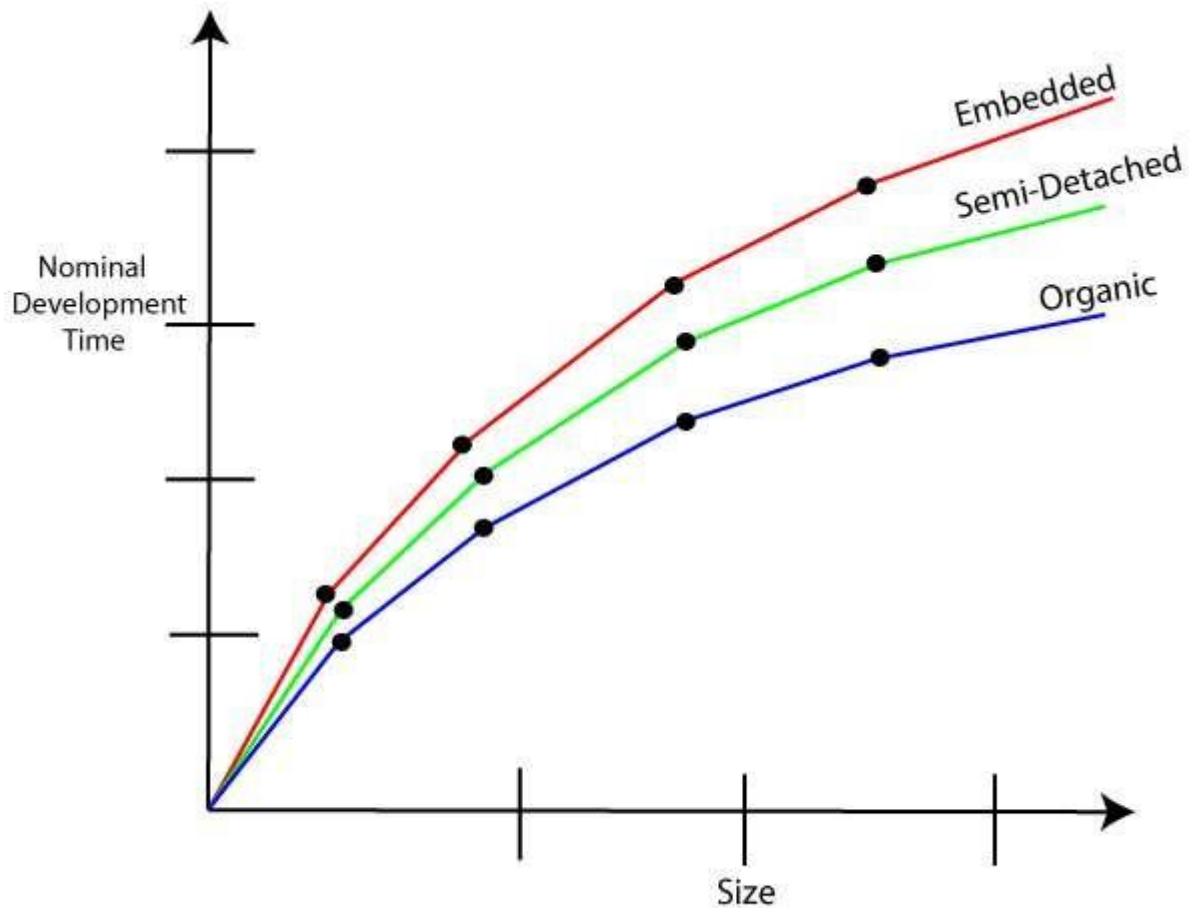
Development time versus size

From the effort estimation, the project cost can be obtained by multiplying the required effort by the manpower cost per month. But, implicit in this project cost computation is the assumption that the entire project cost is incurred on account of the manpower cost alone. In addition to manpower cost, a project would incur costs due to hardware and software required for the project and the company overheads for administration, office space, etc.

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

term nominal implies that if anyone tries to complete the project in a time shorter than the estimated duration, then the cost will increase drastically. But, if anyone completes the project over a longer period of time than the estimated, then there is almost no decrease in the estimated cost value.

Example1: Suppose a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three model i.e., organic, semi-detached & embedded.

Solution: The basic COCOMO equation takes the form:

$$Effort = a_1 * (KLOC) \; a_2 \; PM$$
$$Tdev = b_1 * (efforts) b_2 \; Months$$
Estimated Size of project= 400 KLOC

(i) Organic Mode

$$E = 2.4 * (400)1.05 = 1295.31 \; PM$$
$$D = 2.5 * (1295.31)0.38 = 38.07 \; PM$$

(ii) Semidetached Mode

$$E = 3.0 * (400)1.12 = 2462.79 \; PM$$
$$D = 2.5 * (2462.79)0.35 = 38.45 \; PM$$

(iii) Embedded Mode

$$E = 3.6 * (400)1.20 = 4772.81 \; PM$$
$$D = 2.5 * (4772.8)0.32 = 38 \; PM$$

2. Intermediate Model: The basic Cocomo model considers that the effort is only a function of the number of lines of code and some constants calculated according to the various software systems. The intermediate COCOMO model recognizes these facts and refines the initial estimates obtained through the basic COCOMO model by using a set of 15 cost drivers based on various attributes of software engineering.

Classification of Cost Drivers and their attributes:

(i) Product attributes -

- o   Required software reliability extent
- o   Size of the application database
- o   The complexity of the product

Hardware attributes -

- o   Run-time performance constraints
- o   Memory constraints
- o   The volatility of the virtual machine environment
- o   Required turnabout time

Personnel attributes -

- o   Analyst capability
- o   Software engineering capability
- o   Applications experience
- o   Virtual machine experience
- o   Programming language experience

Project attributes -

- o   Use of software tools
- o   Application of software engineering methods
- o   Required development schedule

3. Detailed COCOMO Model: Detailed COCOMO incorporates all qualities of the standard version with an assessment of the cost driver?s effect on each method of the software engineering process. The detailed model uses various effort multipliers for each cost driver property. In detailed cocomo, the whole software is differentiated into multiple modules, and then we apply COCOMO in various modules to estimate effort and then sum the effort.

The Six phases of detailed COCOMO are:

1. Planning and requirements

2. System structure

3. Complete structure

4. Module code and test

5. Integration and test

6. Cost Constructive model

The effort is determined as a function of program estimate, and a set of cost drivers are given according to every phase of the software lifecycle.

Halstead's Software Metrics

A computer program is an implementation of an algorithm considered to be a collection of tokens which can be classified as either operators or operands. **Halstead's metrics** are included in a number of current commercial tools that count software lines of code. By counting the tokens and determining which operators are and which are operands, the following base measures can be collected:

n1 = Number of distinct operators.

n2 = Number of distinct operands.

N1 = Total number of occurrences of operators.

N2 = Total number of occurrences of operands.

In addition to the above, Halstead defines the following:

n1* = Number of potential operators.

n2* = Number of potential operands.

Halstead refers to n1* and n2* as the minimum possible number of operators and operands for a module and a program respectively. This minimum number would be embodied in the programming language itself, in which the required operation would already exist (for example, in C language, any program must contain at least the definition of the function main()), possibly as a function or as a procedure: n1* = 2, since at least 2 operators must appear for any function or procedure : 1 for the name of the function and 1 to serve as an assignment or grouping symbol, and n2* represents the number of parameters, without repetition, which would need to be passed on to the function or the procedure.

Halstead metrics –

Halstead metrics are:

- **Halstead Program Length –** The total number of operator occurrences and the total number of operand occurrences.

  $N = N1 + N2$

  And estimated program length is, $N^\wedge = n1\log_2 n1 + n2\log_2 n2$

  The following alternate expressions have been published to estimate program length:

  - $N_J = \log_2(n1!) + \log_2(n2!)$
  - $N_B = n1 * \log_2 n2 + n2 * \log_2 n1$
  - $N_C = n1 * sqrt(n1) + n2 * sqrt(n2)$
  - $N_S = (n * \log_2 n) / 2$

- **Halstead Vocabulary –** The total number of unique operator and unique operand occurrences.

  $n = n1 + n2$

- **Program Volume –** Proportional to program size represents the size, in bits, of space necessary for storing the program. This parameter is dependent on specific algorithm implementation. The properties V, N, and the number of lines in the code are shown to be linearly connected and equally valid for measuring relative program size.

  $V = Size * (\log_2 vocabulary) = N * \log_2(n)$

  The unit of measurement of volume is the common unit for size "bits". It is the actual size of a program if a uniform binary encoding for the vocabulary is used. And error = Volume / 3000

- **Potential Minimum Volume –** The potential minimum volume V* is defined as the volume of the most succinct program in which a problem can be coded.

  $V^* = (2 + n2^*) * \log_2(2 + n2^*)$

  Here, n2* is the count of unique input and output parameters

- **Program Level –** To rank the programming languages, the level of abstraction provided by the programming language, Program Level (L) is considered. The higher the level of a language, the less effort it takes to develop a program using that language.

$L = V* / V$

- The value of L ranges between zero and one, with L=1 representing a program written at the highest possible level (i.e., with minimum size). And estimated program level is $L^{\wedge} = 2 * (n2) / (n1)(N2)$

- **Program Difficulty** – This parameter shows how difficult to handle the program is.

  $D = (n1 / 2) * (N2 / n2)$

  $D = 1 / L$

  As the volume of the implementation of a program increases, the program level decreases and the difficulty increases. Thus, programming practices such as redundant usage of operands, or the failure to use higher-level control constructs will tend to increase the volume as well as the difficulty.

- **Programming Effort** – Measures the amount of mental activity needed to translate the existing algorithm into implementation in the specified program language.

  $E = V / L = D * V = Difficulty * Volume$

- **Language Level** – Shows the algorithm implementation program language level. The same algorithm demands additional effort if it is written in a low-level program language. For example, it is easier to program in Pascal than in Assembler.

  $L' = V / D / D$

  $lambda = L * V* = L^2 * V$

- **Intelligence Content** – Determines the amount of intelligence presented (stated) in the program This parameter provides a measurement of program complexity, independently of the program language in which it was implemented.

  $I = V / D$

- **Programming Time** – Shows time (in minutes) needed to translate the existing algorithm into implementation in the specified program language.

  $T = E / (f * S)$

  The concept of the processing rate of the human brain, developed by the psychologist John Stroud, is also used. Stoud defined a moment as the time required by the human brain requires to carry out the most elementary decision. The Stoud number S is therefore Stoud's moments per second with: $5 <= S <= 20$. Halstead uses 18. The value of S has been empirically developed from

psychological reasoning, and its recommended value for programming applications is 18.

Stroud number S = 18 moments / second

seconds-to-minutes factor f = 60

Counting rules for C language –

1. Comments are not considered.
2. The identifier and function declarations are not considered
3. All the variables and constants are considered operands.
4. Global variables used in different modules of the same program are counted asmultiple occurrences of the same variable.
5. Local variables with the same name in different functions are counted as unique operands.
6. Functions calls are considered as operators.
7. All looping statements e.g., do {…} while ( ), while ( ) {…}, for ( ) {…}, all control statements e.g., if ( ) {…}, if ( ) {…} else {…}, etc. are considered as operators.
8. In control construct switch ( ) {case:…}, switch as well as all the case statements are considered as operators.
9. The reserve words like return, default, continue, break, sizeof, etc., are considered as operators.
10. All the brackets, commas, and terminators are considered as operators.
11. GOTO is counted as an operator and the label is counted as an operand.
12. The unary and binary occurrence of "+" and "-" are dealt separately. Similarly "*" (multiplication operator) are dealt separately.
13. In the array variables such as "array-name [index]" "array-name" and "index" are considered as operands and [ ] is considered as operator.
14. In the structure variables such as "struct-name, member-name" or "struct-name -> member-name", struct-name, member-name are taken as operands and '.', '->' are taken as operators. Some names of member elements in different structure variables are counted as unique operands.
15. All the hash directive are ignored.

Example – List out the operators and operands and also calculate the values of software science measures like

```
int sort (int x[ ], int n)


{
   int i, j, save, im1;
   /*This function sorts array x in ascending order */
   If (n< 2) return 1;
   for (i=2; i< =n; i++)
   {
      im1=i-1;
      for (j=1; j< =im1; j++)
         if (x[i] < x[j])
         {
            Save = x[i];
            x[i] = x[j];
            x[j] = save;
         }
   }
   return 0;
}
```

Explanation –

operators	occurrences	operands	occurrences
int	4	sort	1
()	5	x	7
,	4	n	3
[]	7	i	8
if	2	j	7
<	2	save	3
;	11	im1	3
for	2	2	2
=	6	1	3
–	1	0	1
<=	2	–	–
++	2	–	–
return	2	–	–
{}	3	–	–
n1=14	N1=53	n2=10	N2=38

Therefore,

N = 91

n = 24

V = 417.23 bits

N^ = 86.51

n2* = 3 (x:array holding integer

to be sorted. This is used both

as input and output)

V* = 11.6

L = 0.027

D = 37.03

L^ = 0.038

T = 610 seconds

Advantages of Halstead Metrics:
- It is simple to calculate.
- It measures overall quality of the programs.
- It predicts the rate of error.
- It predicts maintenance effort.
- It does not require the full analysis of programming structure.
- It is useful in scheduling and reporting projects.
- It can be used for any programming language.

Disadvantages of Halstead Metrics:

- It depends on the complete code.
- It has no use as a predictive estimating model.

UNIT-4

Requirement Analysis and Specification

IEEE defines requirements analysis as (1) the process of studying user needs to arrive at a definition of a system, hardware or software requirements. (2) The process of studying and refining system, hardware or software requirements.' Requirements analysis helps to understand, interpret, classify, and organize the software requirements in order to assess the feasibility, completeness, and consistency of the requirements.

Requirements gathering: Primarily done during stakeholder meetings, requirements gathering is the exploratory process of researching and documenting project requirements. Shockingly, more than 70 percent of failed projects miss the mark due to a lack of requirements gathering. That no small number.

Truly effective requirements gathering and management is started at the very beginning of the project, and must answer the following questions:

- How long will the project timeline be?

- Who will be involved in the project?

- What are the risks for the requirements gathering process?

- What is our ultimate goal in understanding our project requirements?

It sounds fairly simple, but it's incredibly important.

Why is Requirements Gathering so Important?

Remember back to the last project you managed. What were the risks that came to light? Which resources did you run out of? Was there any scope creep or budgetary mishaps? And overall, what were the impacts of those shortcomings on the project as a whole?

Deadlines, scope, cost overrun—without proper requirements identification at the outset, all of those elements will be affected. Design issues to the product will be impacted, and

developmental delays will occur. Ultimately, your product won't be set up for optimal success as it faces an overrun budget.

The following are some of the well-known requirements gathering techniques −

Brainstorming

Brainstorming is used in requirement gathering to get as many ideas as possible from group of people. Generally used to identify possible solutions to problems, and clarify details of opportunities.

Document Analysis

Reviewing the documentation of an existing system can help when creating AS–IS process document, as well as driving gap analysis for scoping of migration projects. In an ideal world, we would even be reviewing the requirements that drove creation of the existing system – a starting point for documenting current requirements. Nuggets of information are often buried in existing documents that help us ask questions as part of validating requirement completeness.

Focus Group

A focus group is a gathering of people who are representative of the users or customers of a product to get feedback. The feedback can be gathered about needs/opportunities/ problems to identify requirements, or can be gathered to validate and refine already elicited requirements. This form of market research is distinct from brainstorming in that it is a managed process with specific participants.

Interface analysis

Interfaces for a software product can be human or machine. Integration with external systems and devices is just another interface. User centric design approaches are very effective at making sure that we create usable software. Interface analysis – reviewing the touch points with other external systems is important to make sure we don't overlook requirements that aren't immediately visible to users.

Interview

Interviews of stakeholders and users are critical to creating the great software. Without understanding the goals and expectations of the users and stakeholders, we are very unlikely to satisfy them. We also have to recognize the perspective of each interviewee, so that, we can properly weigh and address their inputs. Listening is the skill that helps a great analyst to get more value from an interview than an average analyst.

Observation

By observing users, an analyst can identify a process flow, steps, pain points and opportunities for improvement. Observations can be passive or active (asking questions while observing). Passive observation is better for getting feedback on a prototype (to refine requirements), where active observation is more effective at getting an understanding of an existing business process. Either approach can be used.

Prototyping

Prototyping is a relatively modern technique for gathering requirements. In this approach, you gather preliminary requirements that you use to build an initial version of the solution - a prototype. You show this to the client, who then gives you additional requirements. You change the application and cycle around with the client again. This repetitive process continues until the product meets the critical mass of business needs or for an agreed number of iterations.

Requirement Workshops

Workshops can be very effective for gathering requirements. More structured than a brainstorming session, involved parties collaborate to document requirements. One way to capture the collaboration is with creation of domain-model artifacts (like static diagrams, activity diagrams). A workshop will be more effective with two analysts than with one.

Reverse Engineering When a migration project does not have access to sufficient documentation of the existing system, reverse engineering will identify what the system does. It will not identify what the system should do, and will not identify when the system does the wrong thing.

Survey/Questionnaire

When collecting information from many people – too many to interview with budget and time constraints – a survey or questionnaire can be used. The survey can force users to select from choices, rate something ("Agree Strongly, agree…"), or have open ended questions allowing free-form responses. Survey design is hard – questions can bias the respondents.

Requirements Analysis

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 understand ability 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.

Requirement & Type of requirement

In the software development process, requirement phase is the first software engineering activity. This phase is a user-dominated phase and translates the ideas or views into a requirements document. Note that defining and documenting the user requirements in a concise and unambiguous manner is the first major step to achieve a high-quality product.

The requirement phase encompasses a set of tasks, which help to specify the impact of the software on the organization, customers' needs, and how users will interact with the developed software. The requirements are the basis of the system design. If requirements are not correct the end product will also contain errors. Note that requirements activity like all other software engineering activities should be adapted to the needs of the process, the project, the product and the people involved in the activity. Also, the requirements should be specified at different levels of detail. This is because requirements are meant for people such as users, business managers, system engineers, and so on. For example, business managers are interested in knowing which features can be implemented within the allocated budget whereas end-users are interested in knowing how easy it is to use the features of software.

IEEE defines requirement as (1) A condition or capability needed by a user to solve a problem or achieve an objective. (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents. (3) A documented representation of a condition or capability as in (1) or (2).'

Types of Requirements:

Requirements help to understand the behaviour of a system, which is described by various tasks of the system. For example, some of the tasks of a system are to provide a response to input values, determine the state of data objects, and so on. Note that requirements are considered prior to the development of the software. The requirements, which are commonly considered, are classified into three categories, namely, functional requirements, non-functional requirements, and domain requirements.
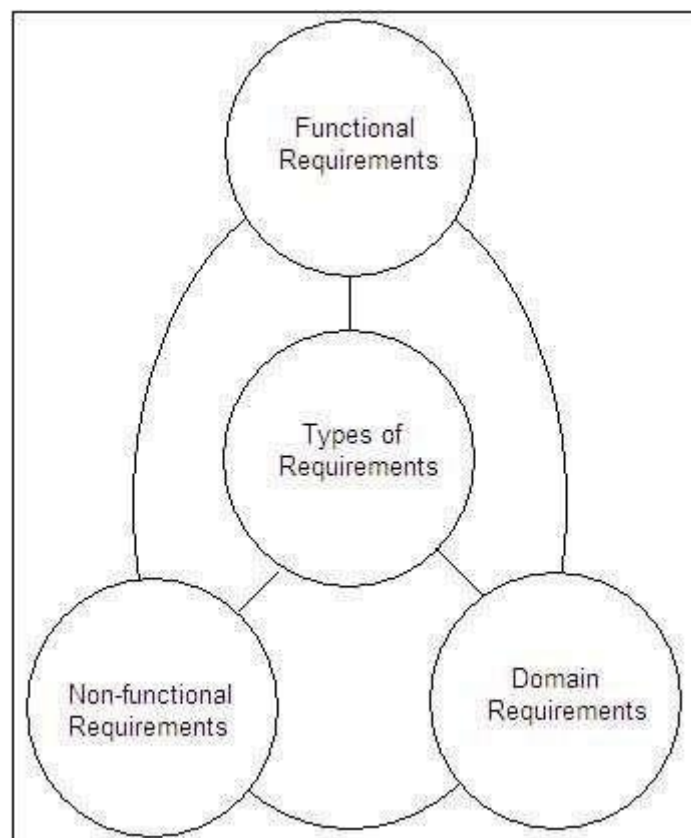
IEEE defines functional requirements as 'a function that a system or component must be able to perform.' These requirements describe the interaction of software with its environment and specify the inputs, outputs, external interfaces, and the functions that should be included in the software. Also, the services provided by functional requirements specify the procedure by which the software should reactto particular inputs or behave in particular situations.

To understand functional requirements properly, let us consider the following example of an online banking system.

1.   The user of the bank should be able to search the desired services from the available ones.

2.   There should be appropriate documents' for users to read. This implies that when a user wants to open an account in the bank, the forms must be available so that the user can open an account.

3.   After registration, the user should be provided with a unique acknowledgement number so that he can later be given an account number.

The above mentioned functional requirements describe the specific services provided by the online banking system. These requirements indicate user requirements and specify that functional requirements may be described at different levels of detail in an online banking system. With the help of these functional requirements, users can easily view, search and download registration forms and other information about the bank. On the other hand, if requirements are not stated properly, they are misinterpreted by software engineers and user requirements are not met.

The functional requirements should be complete and consistent. Completeness implies that all the user requirements are defined. Consistency implies that all requirements are specified clearly without any contradictory definition. Generally, it is observed that completeness and consistency cannot be achieved in large software or in a complex system due to the problems that arise while defining the functional requirements of these systems. The different needs of stakeholders also prevent the achievement of completeness and consistency. Due to these reasons, requirements may not be obvious when they are,'first specified and may further lead to inconsistencies in the requirements specification.

The non-functional requirements (also known as **quality requirements)** are related to system attributes such as reliability and response time. Non-functional requirements arise due to user requirements, budget constraints, organizational policies, and so on. These requirements are not related directly to any particular function provided by the system.

Non-functional requirements should be accomplished in software to make it perform efficiently. For example, if an aeroplane is unable to fulfil reliability requirements, it is not approved for safe operation. Similarly, if a real time control system is ineffective in accomplishing non-functional requirements, the control functions cannot operate correctly.

The description of different types of non-functional requirements is listed below.

1. **Product requirements:** These requirements specify how software product performs. Product requirements comprise the following.

2. **Efficiency requirements:** Describe the extent to which the software makes optimal use of resources, the speed with which the system executes, and the memory it consumes for its operation. For example, the system should be able to operate at least three times faster than the existing system.

3. **Reliability requirements:** Describe the acceptable failure rate of the software. For example, the software should be able to operate even if a hazard occurs.

4. **Portability requirements:** Describe the ease with which the software can be transferred from one platform to another. For example, it should be easy to port the software to a different operating system without the need to redesign the entire software.

5. **Usability requirements:** Describe the ease with which users are able to operate the software. For example, the software should be able to provide access to functionality with fewer keystrokes and mouse clicks.

6. **Organizational requirements:** These requirements are derived from the policies and procedures of an organization. Organizational requirements comprise the following.

7. **Delivery requirements:** Specify when the software and its documentation are to be delivered to the user.

8. **Implementation requirements:** Describe requirements such as programming language and design method.

9. **Standards requirements:** Describe the process standards to be used during software development. For example, the software should be developed using standards specified by the ISO and IEEE standards.

10. **External requirements:** These requirements include all the requirements that affect the software or its development process externally. External requirements comprise the following.

11. **Interoperability requirements:** Define the way in which different computer based systems will interact with each other in one or more organizations.

12. **Ethical requirements:** Specify the rules and regulations of the software so that they are acceptable to users.

13. **Legislative requirements:** Ensure that the software operates within the legal jurisdiction. For example, pirated software should not be sold.
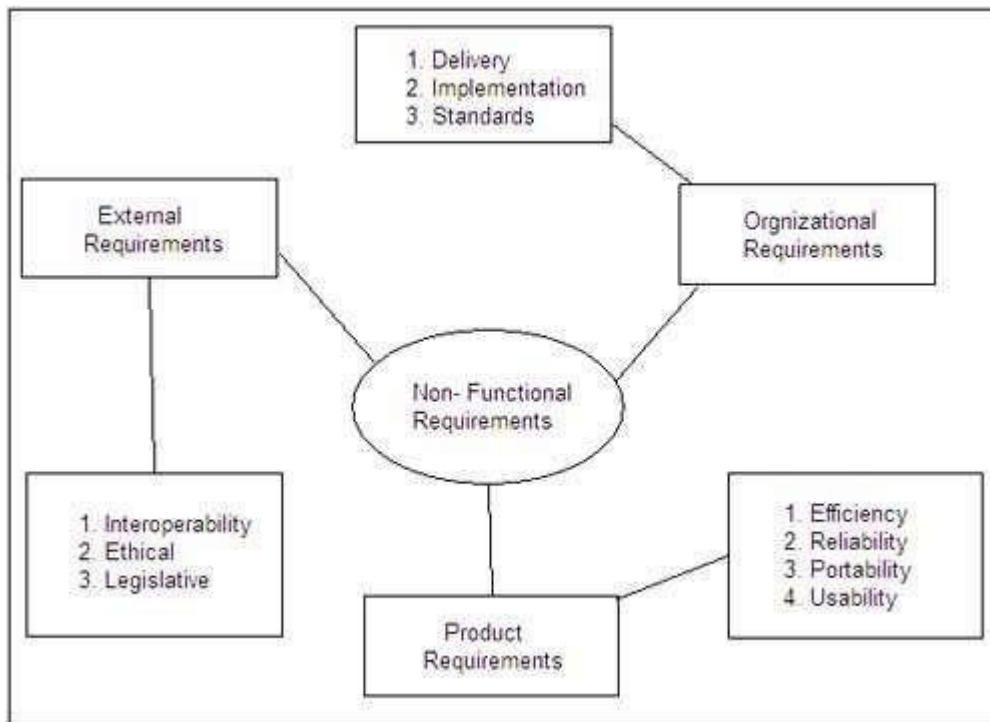
Non-functional requirements are difficult to verify. Hence, it is essential to write non-functional requirements quantitatively, so that they can be tested. For this, non-functional requirements metrics are used.

Software Requirement Specifications

The production of the requirements stage of the software development process is **Software Requirements Specifications (SRS)** (also called a **requirements document**). This report lays a foundation for software engineering activities and is constructing when entire requirements are elicited and analyzed. **SRS** is a formal report, which acts as a representation of software that enables the customers to review whether it (SRS) is according to their requirements. Also, it comprises user requirements for a system as well as detailed specifications of the system requirements.

The SRS is a specification for a specific software product, program, or set of applications that perform particular functions in a specific environment. It serves several goals depending on who is writing it. First, the SRS could be written by the client of a system. Second, the SRS could be written by a developer of the system. The two methods create entirely various situations and establish different purposes for the document altogether. The first case, SRS, is

used to define the needs and expectation of the users. The second case, SRS, is written for various purposes and serves as a contract document between customer and developer.

Characteristics of good SRS

Following are the features of a good SRS document:

1. Correctness: User review is used to provide the accuracy of requirements stated in the SRS. SRS is said to be perfect if it covers all the needs that are truly expected from the system.

2. Completeness: The SRS is complete if, and only if, it includes the following elements:

(1). All essential requirements, whether relating to functionality, performance, design, constraints, attributes, or external interfaces.

(2). Definition of their responses of the software to all realizable classes of input data in all available categories of situations.

(3). Full labels and references to all figures, tables, and diagrams in the SRS and definitions of all terms and units of measure.

3. Consistency: The SRS is consistent if, and only if, no subset of individual requirements described in its conflict. There are three types of possible conflict in the SRS:

(1). The specified characteristics of real-world objects may conflicts. For example,

(a) The format of an output report may be described in one requirement as tabular but in another as textual.

(b) One condition may state that all lights shall be green while another states that all lights shall be blue.

(2). There may be a reasonable or temporal conflict between the two specified actions. For example,

(a) One requirement may determine that the program will add two inputs, and another may determine that the program will multiply them.

(b) One condition may state that "A" must always follow "B," while other requires that "A and B" co-occurs.

(3). Two or more requirements may define the same real-world object but use different terms for that object. For example, a program's request for user input may be called a "prompt" in one requirement's and a "cue" in another. The use of standard terminology and descriptions promotes consistency.

4. Unambiguousness: SRS is unambiguous when every fixed requirement has only one interpretation. This suggests that each element is uniquely interpreted. In case there is a method used with multiple definitions, the requirements report should determine the implications in the SRS so that it is clear and simple to understand.

5. Ranking for importance and stability: The SRS is ranked for importance and stability if each requirement in it has an identifier to indicate either the significance or stability of that particular requirement.

Typically, all requirements are not equally important. Some prerequisites may be essential, especially for life-critical applications, while others may be desirable. Each element should be identified to make these differences clear and explicit. Another way to rank requirements is to distinguish classes of items as essential, conditional, and optional.

6. Modifiability: SRS should be made as modifiable as likely and should be capable of quickly obtain changes to the system to some extent. Modifications should be perfectly indexed and cross-referenced.

7. Verifiability: SRS is correct when the specified requirements can be verified with a cost-effective system to check whether the final software meets those requirements. The requirements are verified with the help of reviews.

8. Traceability: The SRS is traceable if the origin of each of the requirements is clear and if it facilitates the referencing of each condition in future development or enhancement documentation.

There are two types of Traceability:

1. Backward Traceability: This depends upon each requirement explicitly referencing its source in earlier documents.

2. Forward Traceability: This depends upon each element in the SRS having a unique name or reference number.

The forward traceability of the SRS is especially crucial when the software product enters the operation and maintenance phase. As code and design document is modified, it is necessary to be able to ascertain the complete set of requirements that may be concerned by those modifications.

9. Design Independence: There should be an option to select from multiple design alternatives for the final system. More specifically, the SRS should not contain any implementation details.

10. Testability: An SRS should be written in such a method that it is simple to generate test cases and test plans from the report.

11. Understandable by the customer: An end user may be an expert in his/her explicit domain but might not be trained in computer science. Hence, the purpose of formal notations and symbols should be avoided too as much extent as possible. The language should be kept simple and clear.

12. The right level of abstraction: If the SRS is written for the requirements stage, the details should be explained explicitly. Whereas,for a feasibility study, fewer analysis can be used. Hence, the level of abstraction modifies according to the objective of the SRS.

Properties of a good SRS document

The essential properties of a good SRS document are the following:

Concise: The SRS report should be concise and at the same time, unambiguous, consistent, and complete. Verbose and irrelevant descriptions decrease readability and also increase error possibilities.

Structured: It should be well-structured. A well-structured document is simple to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the user requirements. Often, user requirements evolve over a period of time. Therefore, to make the modifications to the SRS document easy, it is vital to make the report well-structured.

Black-box view: It should only define what the system should do and refrain from stating how to do these. This means that the SRS document should define the external behaviour of the system and not discuss the implementation issues. The SRS report should view the system to be developed as a black box and should define the externally visible behaviour of the system. For this reason, the SRS report is also known as the black-box specification of a system.

Conceptual integrity: It should show conceptual integrity so that the reader can merely understand it. Response to undesired events: It should characterize acceptable responses to unwanted events. These are called system response to exceptional conditions.

Verifiable: All requirements of the system, as documented in the SRS document, should be correct. This means that it should be possible to decide whether or not requirements have been met in an implementation.

UNIT-5

Software Design and Implementation

Software design and implementation is the stage in the software engineering process at which an executable software system is developed. • Software design and implementation activities are invariably inter-leaved. – Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements. – Implementation is the process of realizing the design as a program.

Characteristics of a good software design

For good quality software to be produced, the software design must also be of good quality. Now, the matter of concern is how the quality of good software design is measured? This is done by observing certain factors in software design. These factors are:

1. Correctness
2. Understandability
3. Efficiency
4. Maintainability

Now, let us define each of them in detail,

1) Correctness

First of all, the design of any software is evaluated for its correctness. The evaluators check the software for every kind of input and action and observe the results that the software will produce according to the proposed design. If the results are correct for every input, the design is accepted and is considered that the software produced according to this design will function correctly.

2) Understandability

The software design should be understandable so that the developers do not find any difficulty to understand it. Good software design should be self- explanatory. This is because there are hundreds and thousands of developers that develop different modules of the

software, and it would be very time consuming to explain each design to each developer. So, if the design is easy and self- explanatory, it would be easy for the developers to implement it and build the same software that is represented in the design.

3) Efficiency

The software design must be efficient. The efficiency of the software can be estimated from the design phase itself, because if the design is describing software that is not efficient and useful, then the developed software would also stand on the same level of efficiency. Hence, for efficient and good quality software to be developed, care must be taken in the designing phase itself.

4) Maintainability

The software design must be in such a way that modifications can be easily made in it. This is because every software needs time to time modifications and maintenance. So, the design of the software must also be able to bear such changes. It should not be the case that after making some modifications the other features of the software start misbehaving. Any change made in the software design must not affect the other available features, and if the features are getting affected, then they must be handled properly.

Coupling and Cohesion

Module Coupling

In software engineering, the coupling is the degree of interdependence between software modules. Two modules that are tightly coupled are strongly dependent on each other. However, two modules that are loosely coupled are not dependent on each other. **Uncoupled modules** have no interdependence at all within them.

The various types of coupling techniques are shown in fig:

Module Coupling

Uncoupled: no dependencies
(a)

Loosely Coupled: Some dependencies
(b)

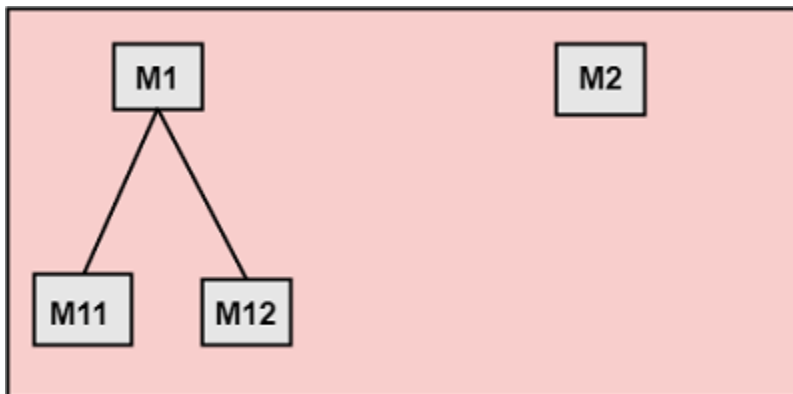Highly Coupled: Many dependencies
(c)
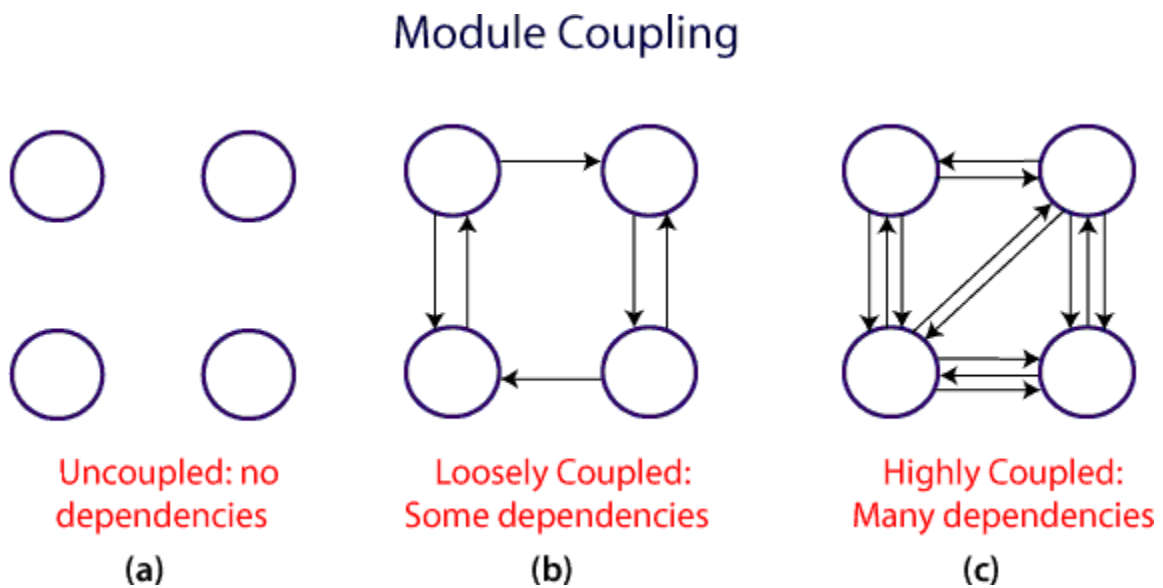
A good design is the one that has low coupling. Coupling is measured by the number of relations between the modules. That is, the coupling increases as the number of calls between modules increase or the amount of shared data is large. Thus, it can be said that a design with high coupling will have more errors.

Types of Module Coupling

Types of Modules Coupling

There are various types of module Coupling are as follows:

No Direct Coupling

Data Coupling

Stamp Coupling

Control Coupling

External Coupling

Common Coupling

Content Coupling

Best

Worst
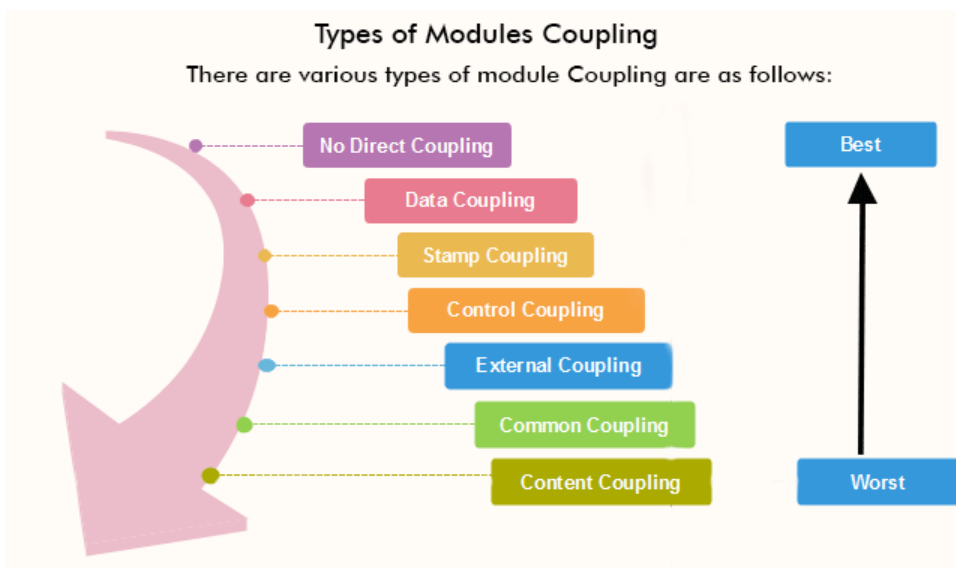
1. No Direct Coupling: There is no direct coupling between M1 and M2.

In this case, modules are subordinates to different modules. Therefore, no direct coupling.

2. Data Coupling: When data of one module is passed to another module, this is called data coupling.
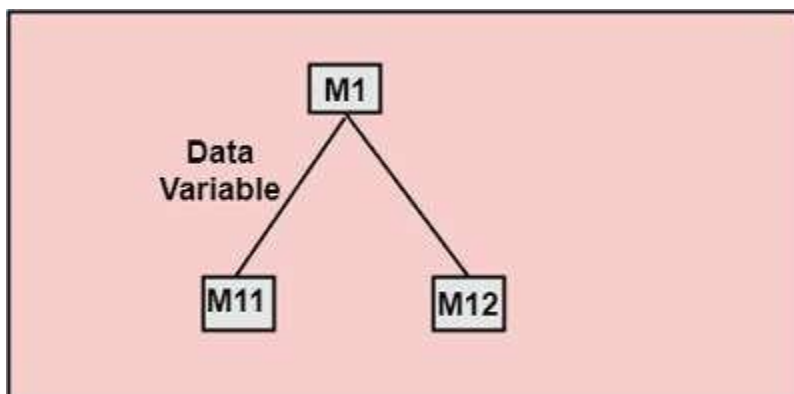
3. Stamp Coupling: Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.

4. Control Coupling: Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.

5. External Coupling: External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.

6. Common Coupling: Two modules are common coupled if they share information through some global data items.
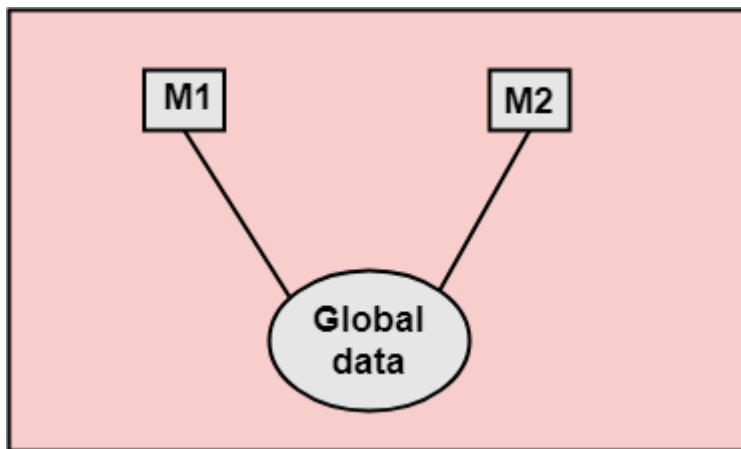

7. Content Coupling: Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.
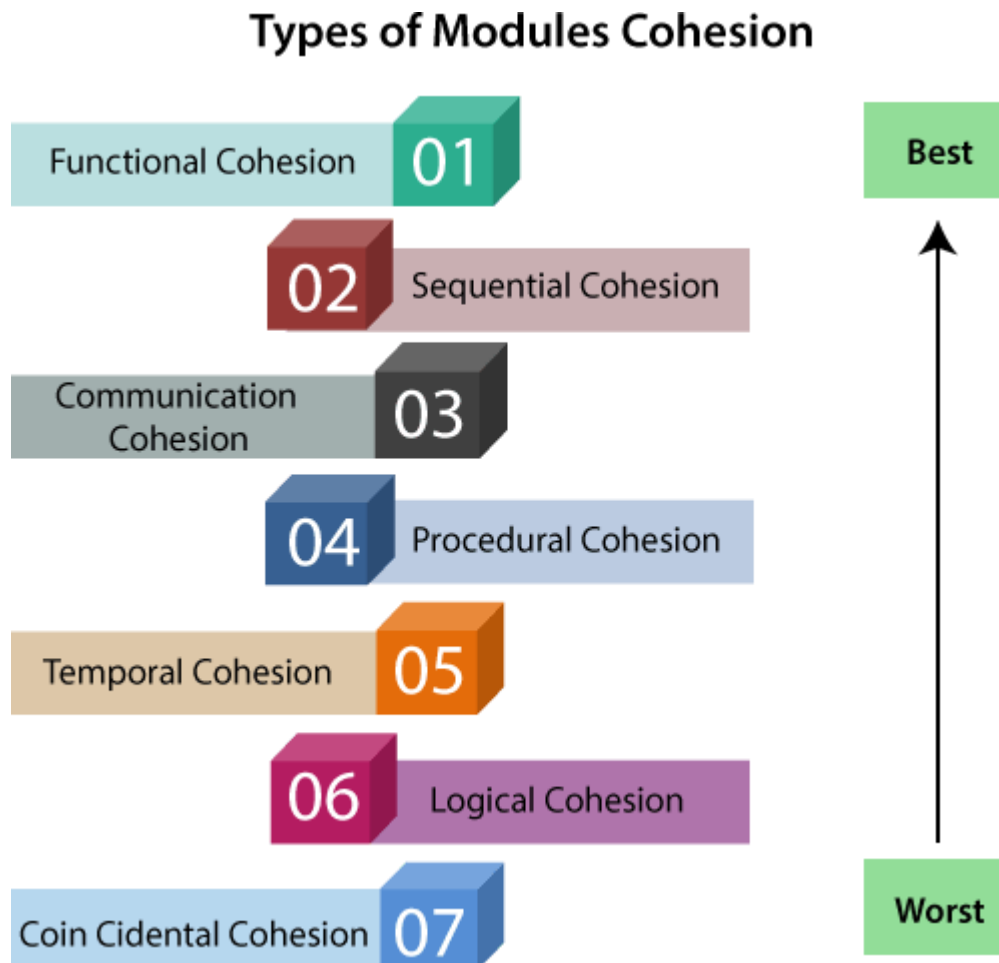

Module Cohesion

In computer programming, cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationships between pieces of functionality within a given module. For example, in highly cohesive systems, functionality is strongly related.

Cohesion is an **ordinal** type of measurement and is generally described as "high cohesion" or "low cohesion."
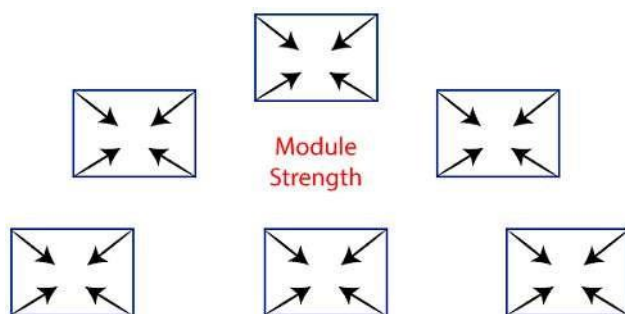
Cohesion= Strength of relations within Modules

Types of Modules Cohesion

Types of Modules Cohesion

Functional Cohesion	**01**		**Best**
	02 Sequential Cohesion		
Communication Cohesion	**03**		
	04 Procedural Cohesion		
Temporal Cohesion	**05**		
	06 Logical Cohesion		
Coin Cidental Cohesion	**07**		**Worst**

1. **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.

2. **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.

3. **Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.

4. **Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.

5. **Temporal Cohesion:** When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.

6. **Logical Cohesion:** A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.

7. **Coincidental Cohesion:** A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all.

Differentiate between Coupling and Cohesion

| Coupling	Cohesion
Coupling is also called Inter-Module Binding.	Cohesion is also called Intra-Module Binding.
Coupling shows the relationships between modules.	Cohesion shows the relationship within the module.
Coupling shows the relative **independence** between the modules.	Cohesion shows the module's relative **functional** strength.
While creating, you should aim for low coupling, i.e., dependency among modules should be less.	While creating you should aim for high cohesion, i.e., a cohesive component/ module focuses on a single function (i.e., single-mindedness) with little interaction with other modules of the system.
In coupling, modules are linked to the other modules.	In cohesion, the module focuses on a single thing.

Function Oriented Design

Function Oriented design is a method to software design where the model is decomposed into a set of interacting units or modules where each unit or module has a clearly defined function. Thus, the system is designed from a functional viewpoint.

Design Notations

Design Notations are primarily meant to be used during the process of design and are used to represent design or design decisions. For a function-oriented design, the design can be represented graphically or mathematically by the following:
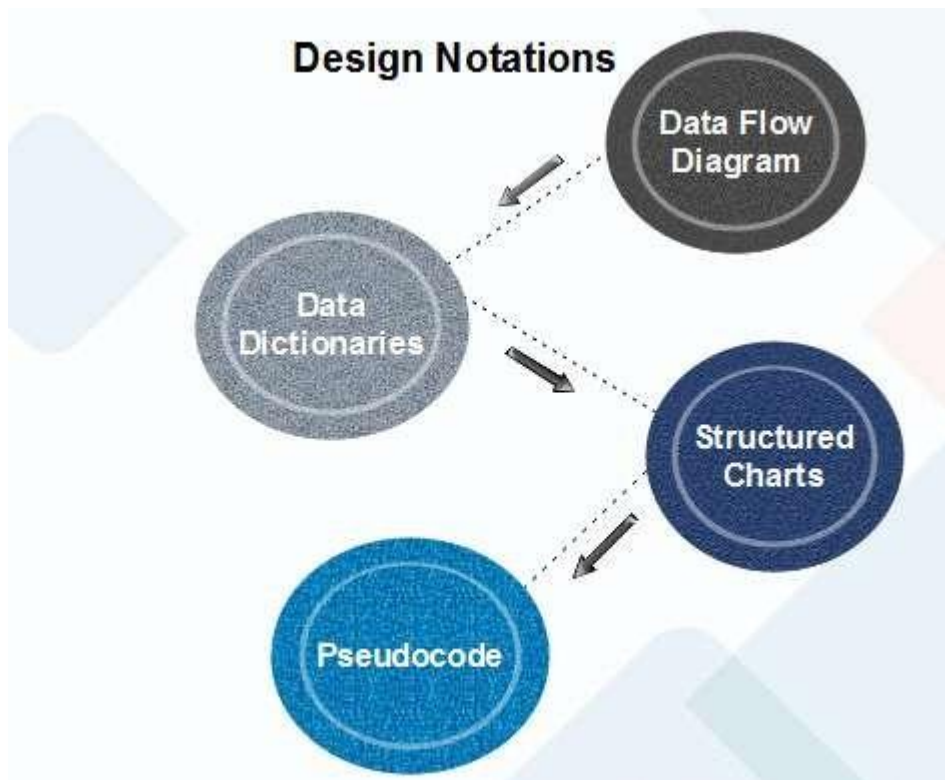
Data Flow Diagram

Data-flow design is concerned with designing a series of functional transformations that convert system inputs into the required outputs. The design is described as data-flow

diagrams. These diagrams show how data flows through a system and how the output is derived from the input through a series of functional transformations.

Data-flow diagrams are a useful and intuitive way of describing a system. They are generally understandable without specialized training, notably if control information is excluded. They show end-to-end processing. That is the flow of processing from when data enters the system to where it leaves the system can be traced.

Data-flow design is an integral part of several design methods, and most CASE tools support data-flow diagram creation. Different ways may use different icons to represent data-flow diagram entities, but their meanings are similar.

The notation which is used is based on the following symbols:

Symbol	Name	Meaning
	Rounded Rectangle	It represents functions which transforms input to output. The transformation name indicates its function.
	Rectangle	It represents data stores. Again, they should give a descriptive name.
	Circle	It represents user interactions with the system that provides input or receives output.
→	Arrows	It shows the direction of data flow. Their name describes the data flowing along the path.
"and" and "or"	Keywords	The keywords "and" and "or". These have their usual meanings in boolean expressions. They are used to link data flows when more than one data flow may be input or output from a transformation.

Data flow diagram of a design report generator
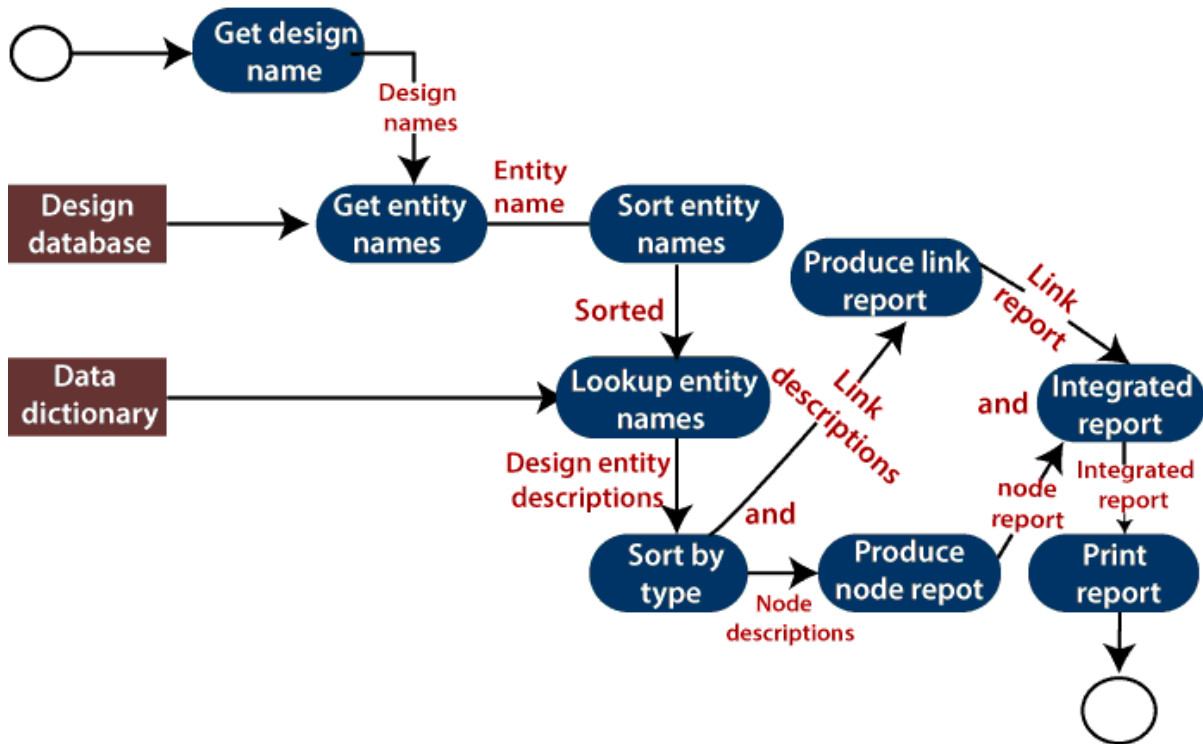
The report generator produces a report which describes all of the named entities in a data-flow diagram. The user inputs the name of the design represented by the diagram. The report generator then finds all the names used in the data-flow diagram. It looks up a data dictionary and retrieves information about each name. This is then collated into a report which is output by the system.

Data Dictionaries

A data dictionary lists all data elements appearing in the DFD model of a system. The data items listed contain all data flows and the contents of all data stores looking on the DFDs in the DFD model of a system.

A data dictionary lists the objective of all data items and the definition of all composite data elements in terms of their component data items. For example, a data dictionary entry may contain that the data *grossPay* consists of the parts *regularPay* and *overtimePay*.

grossPay = regularPay + overtimePay

For the smallest units of data elements, the data dictionary lists their name and their type.

A data dictionary plays a significant role in any software development process because of the following reasons:

- A Data dictionary provides a standard language for all relevant information for use by engineers working in a project. A consistent vocabulary for data items is essential since, in large projects, different engineers of the project tend to use different terms to refer to the same data, which unnecessarily causes confusion.

- The data dictionary provides the analyst with a means to determine the definition of various data structures in terms of their component elements.
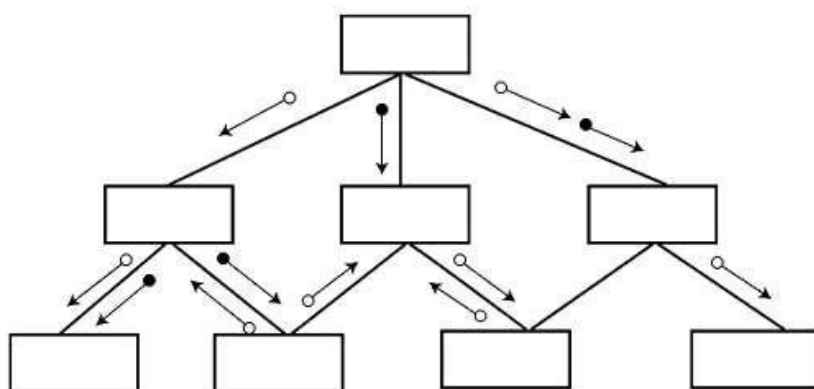
Structured Charts

It partitions a system into block boxes. A Black box system that functionality is known to the user without the knowledge of internal design.

Structured Chart is a graphical representation which shows:

- System partitions into modules
- Hierarchy of component modules
- The relation between processing modules
- Interaction between modules
- Information passed between modules

The following notations are used in structured chart:

Hierarchical format of a structure chart

Structured Chart is a graphical representation which shows:

- System partitions into modules
- Hierarchy of component modules
- The relation between processing modules
- Interaction between modules
- Information passed between modules

The following notations are used in structured chart:

SYMBOL	DESCRIPTION
	Module
	Arrow
	Data couple
	Control Flag
	Loop
	Decision

Pseudo-code

Pseudo-code notations can be used in both the preliminary and detailed design phases. Using pseudo-code, the designer describes system characteristics using short, concise, English Language phases that are structured by keywords such as If-Then-Else, While-Do, and End.

<h1 style="text-align:center">Object-Oriented Design</h1>

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

The different terms related to object design are:
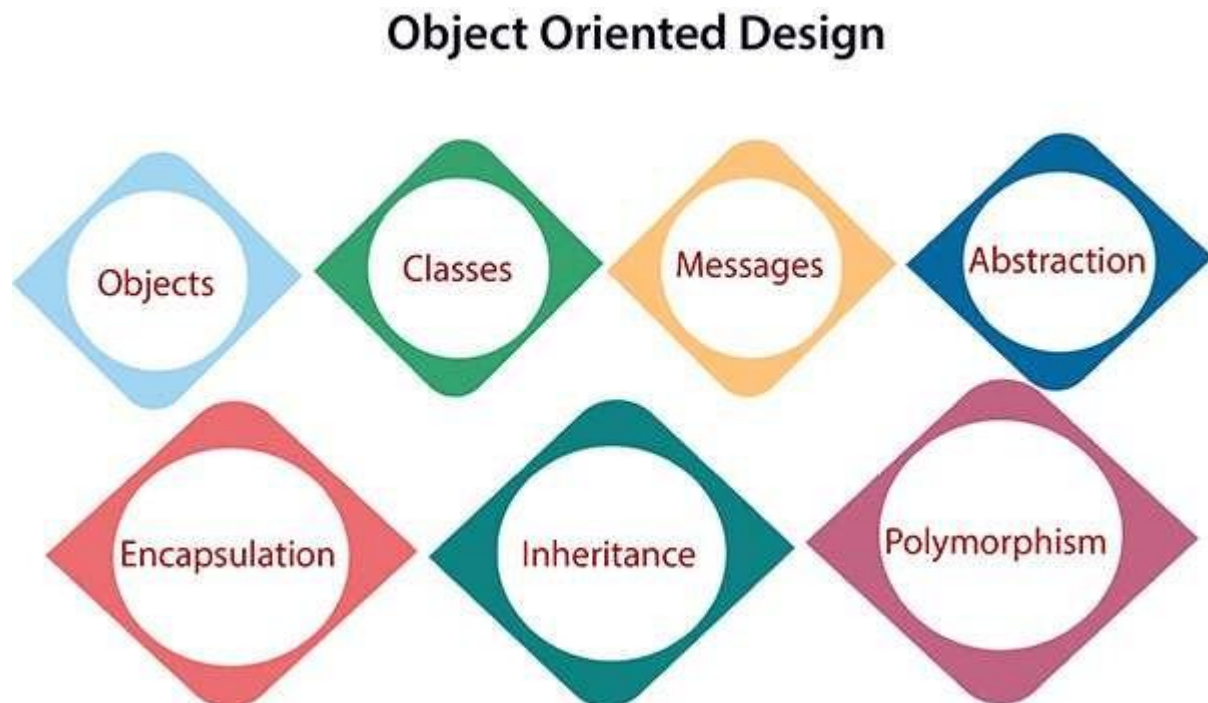
1. **Objects:** All entities involved in the solution design are known as objects. For example, person, banks, company, and users are considered as objects. Every entity has some attributes associated with it and has some methods to perform on the attributes.

2. **Classes:** A class is a generalized description of an object. An object is an instance of a class. A class defines all the attributes, which an object can have and methods, which represents the functionality of the object.

3. **Messages:** Objects communicate by message passing. Messages consist of the integrity of the target object, the name of the requested operation, and any other action needed to perform the function. Messages are often implemented as procedure or function calls.

4. **Abstraction** In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials.

5. **Encapsulation:** Encapsulation is also called an information hiding concept. The data and operations are linked to a single unit. Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.

6. **Inheritance:** OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate superclasses.This property of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.

7. **Polymorphism:** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name. This is known as polymorphism, which allows a single interface is performing functions for different types. Depending upon how the service is invoked, the respective portion of the code gets executed.

Structured Coding Techniques

Structured Programming

In the process of coding, the lines of code keep multiplying, thus, size of the software increases. Gradually, it becomes next to impossible to remember the flow of program. If one forgets how software and its underlying programs, files, procedures are constructed it then becomes very difficult to share, debug and modify the program. The solution to this is structured programming. It encourages the developer to use subroutines and loops instead of using simple jumps in the code, thereby bringing clarity in the code and improving its efficiency Structured programming also helps programmer to reduce coding time and organize code properly.

Structured programming states how the program shall be coded. Structured programming uses three main concepts:

- **Top-down analysis** - software is always made to perform some rational work. This rational work is known as problem in the software parlance. Thus it is very important that we understand how to solve the problem. Under top-down analysis, the problem is broken down into small pieces where each one has some significance. Each problem is individually solved and steps are clearly stated about how to solve the problem.

- **Modular Programming** - While programming, the code is broken down into smaller group of instructions. These groups are known as modules, subprograms or subroutines. Modular programming based on the understanding of top-down analysis. It discourages jumps using 'goto' statements in the program, which often makes the program flow non-traceable. Jumps are prohibited and modular format is encouraged in structured programming.

- **Structured Coding** - In reference with top-down analysis, structured coding sub-divides the modules into further smaller units of code in the order of their execution. Structured programming uses control structure, which controls the flow of the program, whereas structured coding uses control structure to organize its instructions in definable patterns.

Programming style

Programming style is set of coding rules followed by all the programmers to write the code. When multiple programmers work on the same software project, they frequently need to work with the program code written by some other developer. This becomes tedious or at times impossible, if all developers do not follow some standard programming style to code the program.

An appropriate programming style includes using function and variable names relevant to the intended task, using well-placed indentation, commenting code for the convenience of reader and overall presentation of code. This makes the program code readable and understandable by all, which in turn makes debugging and error solving easier. Also, proper coding style helps ease the documentation and updation.

Coding Guidelines

General coding guidelines provide the programmer with a set of the best methods which can be used to make programs more comfortable to read and maintain. Most of the examples use the C language syntax, but the guidelines can be tested to all languages.

The following are some representative coding guidelines recommended by many software development organizations.

1. Line Length: It is considered a good practice to keep the length of source code lines at or below 80 characters. Lines longer than this may not be visible properly on some terminals and tools. Some printers will truncate lines longer than 80 columns.

2. Spacing: The appropriate use of spaces within a line of code can improve readability.

Example:

Bad: cost=price+(price*sales_tax)
          fprintf(stdout ,"The total cost is %5.2f\n",cost);

Better: cost = price + (price * sales_tax)
          fprintf (stdout,"The total cost is %5.2f\n",cost);

3. The code should be well-documented: As a rule of thumb, there must be at least one comment line on the average for every three-source line.

4. The length of any function should not exceed 10 source lines: A very lengthy function is generally very difficult to understand as it possibly carries out many various functions. For the same reason, lengthy functions are possible to have a disproportionately larger number of bugs.

5. Do not use goto statements: Use of goto statements makes a program unstructured and very tough to understand.

6. Inline Comments: Inline comments promote readability.

7. Error Messages: Error handling is an essential aspect of computer programming. This does not only include adding the necessary logic to test for and handle errors but also involves making error messages meaningful.

8. Clarity and simplicity of Expression: The programs should be designed in such a manner so that the objectives of the program is clear.

9. Naming: In a program, you are required to name the module, processes, and variable, and so on. Care should be taken that the naming style should not be cryptic and non-representative.

    For Example: a = 3.14 * r * r

                area of circle = 3.14 * radius * radius;

10. Control Constructs: It is desirable that as much as a possible single entry and single exit constructs used.

11. Information hiding: The information secure in the data structures should be hidden from the rest of the system where possible. Information hiding can decrease the coupling between modules and make the system more maintainable.

12. Nesting: Deep nesting of loops and conditions greatly harm the static and dynamic behavior of a program. It also becomes difficult to understand the program logic, so it is desirable to avoid deep nesting.

13. User-defined types: Make heavy use of user-defined data types like enum, class, structure, and union. These data types make your program code easy to write and easy to understand.

14. Module size: The module size should be uniform. The size of the module should not be too big or too small. If the module size is too large, it is not generally functionally cohesive. If the module size is too small, it leads to unnecessary overheads.

15. Module Interface: A module with a complex interface should be carefully examined.

16. Side-effects: When a module is invoked, it sometimes has a side effect of modifying the program state. Such side-effect should be avoided where as possible.

Software Documentation

Software documentation is an important part of software process. A well written document provides a great tool and means of information repository necessary to know about software process. Software documentation also provides information about how to use the product.

A well-maintained documentation should involve the following documents:

- **Requirement documentation** - This documentation works as key tool for software designer, developer and the test team to carry out their respective tasks. This document contains all the functional, non-functional and behavioural description of the intended software.

  Source of this document can be previously stored data about the software, already running software at the client's end, client's interview, questionnaires and research. Generally it is stored in the form of spreadsheet or word processing document with the high-end software management team.

  This documentation works as foundation for the software to be developed and is majorly used in verification and validation phases. Most test-cases are built directly from requirement documentation.

- **Software Design documentation** - These documentations contain all the necessary information, which are needed to build the software. It contains: **(a)** High-level software architecture, **(b)** Software design details, **(c)** Data flow diagrams, **(d)** Database design

  These documents work as repository for developers to implement the software. Though these documents do not give any details on how to code the program, they give all necessary information that is required for coding and implementation.

- **Technical documentation** - These documentations are maintained by the developers and actual coders. These documents, as a whole, represent information about the code. While writing the code, the programmers also mention objective of the code, who wrote it, where will it be required, what it does and how it does, what other resources the code uses, etc.

The technical documentation increases the understanding between various programmers working on the same code. It enhances re-use capability of the code. It makes debugging easy and traceable.

There are various automated tools available and some comes with the programming language itself. For example java comes JavaDoc tool to generate technical documentation of code.

- **User documentation** - This documentation is different from all the above explained. All previous documentations are maintained to provide information about the software and its development process. But user documentation explains how the software product should work and how it should be used to get the desired results.

  These documentations may include, software installation procedures, how-to guides, user-guides, uninstallation method and special references to get more information like license updation etc.

UNIT-6

SOFTWARE TESTING is defined as an activity to check whether the actual results match the expected results and to ensure that the software system is Defect free. It involves the execution of a software component or system component to evaluate one or more properties of interest. Software testing also helps to identify errors, gaps, or missing requirements in contrary to the actual requirements. It can be either done manually or using automated.

Verification and Validation

Verification and Validation is the process of investigating that a software system satisfies specifications and standards and it fulfills the required purpose. **Barry Boehm** described verification and validation as the following:

Verification: Are we building the product right?
Validation: Are we building the right product?

Verification:

Verification is the process of checking that a software achieves its goal without any bugs. It is the process to ensure whether the product that is developed is right or not. It verifies whether the developed product fulfills the requirements that we have. Verification is **Static Testing**.

Activities involved in verification:

1. Inspections
2. Reviews
3. Walkthroughs
4. Desk-checking

Validation:

Validation is the process of checking whether the software product is up to the mark or in other words product has high level requirements. It is the process of checking the validation of product i.e. it checks what we are developing is the right product. it is validation of actual and expected product.
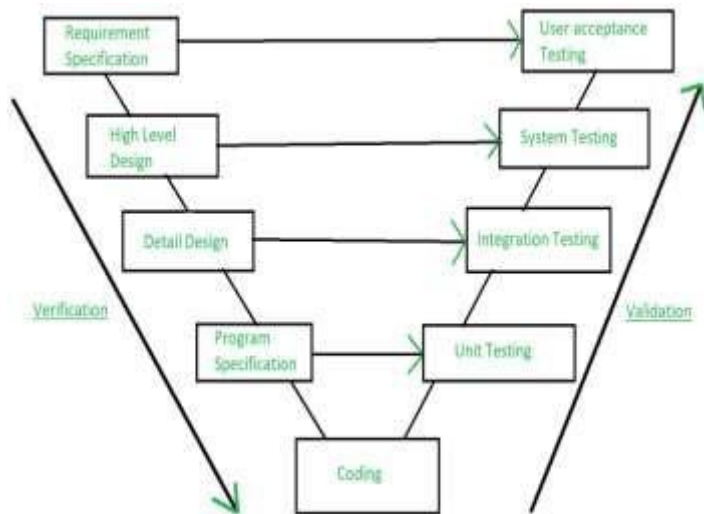
Validation is the **Dynamic Testing**.

Activities involved in validation:

1. Black box testing
2. White box testing

3. Unit testing
4. Integration testing

Note: Verification is followed by Validation.

Types of Software Testing

Introduction:-

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

Principles of Testing:-

(i) All the test should meet the customer requirements

(ii) To make our software testing should be performed by third party

(iii) Exhaustive testing is not possible. As we need the optimal amount of testing based on the risk assessment of the application.

(iv) All the test to be conducted should be planned before implementing it

(v) It follows pare to rule(80/20 rule) which states that 80% of errors comes from 20% of program components.

(vi) Start testing with small parts and extend it to large parts.

Types of Testing:-

1. Unit Testing

It focuses on smallest unit of software design. In this we test an individual unit or group of inter related units. It is often done by programmer by using sample input and observing its corresponding outputs.

Example:

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

   Function is working fine

b) Misunderstood or incorrect, arithmetic precedence.

c) Incorrect initialization

2. Integration Testing

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

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

3. Regression Testing

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

Example

In school record suppose we have module staff, students

and finance combining these modules and checking if on

integration these module works fine is regression testing

4. Smoke Testing

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

If project has 2 modules so before going to module

make sure that module 1 works properly

5. Alpha Testing

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

When software testing is performed internally within the organization

6. Beta Testing

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

When software testing is performed for the limited number of people

7. System Testing

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

This include functional as well as non functional testing

8. Stress Testing

In this we gives unfavourable conditions to the system and check how they perform in those condition.

Example:

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

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

(c) Test cases that may cause excessive disk requirement

9. Performance Testing

It is designed to test the run-time performance of software within the context of an integrated system. It is used to test speed and effectiveness of program. Example:

Checking number of processor cycles.

Black box testing

Black box testing is a type of software testing in which the functionality of the software is not known. The testing is done without the internal knowledge of the products.

Black box testing can be done in following ways:

1. Syntax Driven Testing – This type of testing is applied to systems that can be syntactically represented by some language. For example- compilers, language that can be represented by context free grammar. In this, the test cases are generated so that each grammar rule is used at least once.

2. Equivalence partitioning – It is often seen that many type of inputs work similarly so instead of giving all of them separately we can group them together and test only one input of each group. The idea is to partition the input domain of the system into a number of equivalence classes such that each member of class works in a similar way, i.e., if a test case in one class results in some error, other members of class would also result into same error. The technique involves two steps:

1. **Identification of equivalence class** – Partition any input domain into minimum two sets: **valid values** and **invalid values**. For example, if the valid range is 0 to 100 then select one valid input like 49 and one invalid like 104.

2. **Generating test cases –**

    (i) To each valid and invalid class of input assign unique identification number.

    (ii) Write test case covering all valid and invalid test case considering that no two invalid inputs mask each other.

1. To calculate the square root of a number, the equivalence classes will be:

    (a) Valid inputs:
    - Whole number which is a perfect square- output will be an integer.
    - Whole number which is not a perfect square- output will be decimal number.
    - Positive decimals

    (b) Invalid inputs:
    - Negative numbers (integer or decimal).
    - Characters other that numbers like "a","!",";",etc.

3. **Boundary value analysis** – Boundaries are very good places for errors to occur. Hence if test cases are designed for boundary values of input domain then the efficiency of testing improves and probability of finding errors also increases. For example – If valid range is 10 to 100 then test for 10,100 also apart from valid and invalid inputs.

4. **Cause effect Graphing** – This technique establishes relationship between logical input called causes with corresponding actions called effect. The causes and effects are represented using Boolean graphs. The following steps are followed:

1. Identify inputs (causes) and outputs (effect).
2. Develop cause effect graph.
3. Transform the graph into decision table.
4. Convert decision table rules to test cases.

For example, in the following cause effect graph:

It can be converted into decision table like:

		1	2	3	4
CAUSES	C1	1	0	0	0
	C2	0	1	0	0
	C3	0	0	1	1
	C4	1	0	0	0
	C5	0	1	1	0
	C6	0	0	0	1
EFFECTS	E1	X	-	-	-
	E2	-	X	-	-
	E3	-	-	X	-
	E4	-	-	-	X

Each column corresponds to a rule which will become a test case for testing. So there will be 4 test cases.

5. **Requirement based testing** – It includes validating the requirements given in SRS of software system.

6. **Compatibility testing** – The test case result not only depend on product but also infrastructure for delivering functionality. When the infrastructure parameters are changed it is still expected to work properly. Some parameters that generally affect compatibility of software are:

1. Processor (Pentium 3,Pentium 4) and number of processors.
2. Architecture and characteristic of machine (32 bit or 64 bit).
3. Back-end components such as database servers.
4. Operating System (Windows, Linux, etc).

White box Testing

White box testing techniques analyze the internal structures the used data structures, internal design, code structure and the working of the software rather than just the functionality as in black box testing. It is also called glass box testing or clear box testing or structural testing.

Working process of white box testing:

- **Input:** Requirements, Functional specifications, design documents, source code.
- **Processing:** Performing risk analysis for guiding through the entire process.
- **Proper test planning:** Designing test cases so as to cover entire code. Execute rinse-repeat until error-free software is reached. Also, the results are communicated.

- **Output:** Preparing final report of the entire testing process.

Testing techniques:

- **Statement coverage:** In this technique, the aim is to traverse all statement at least once. Hence, each line of code is tested. In case of a flowchart, every node must be traversed at least once. Since all lines of code are covered, helps in pointing out faulty code.
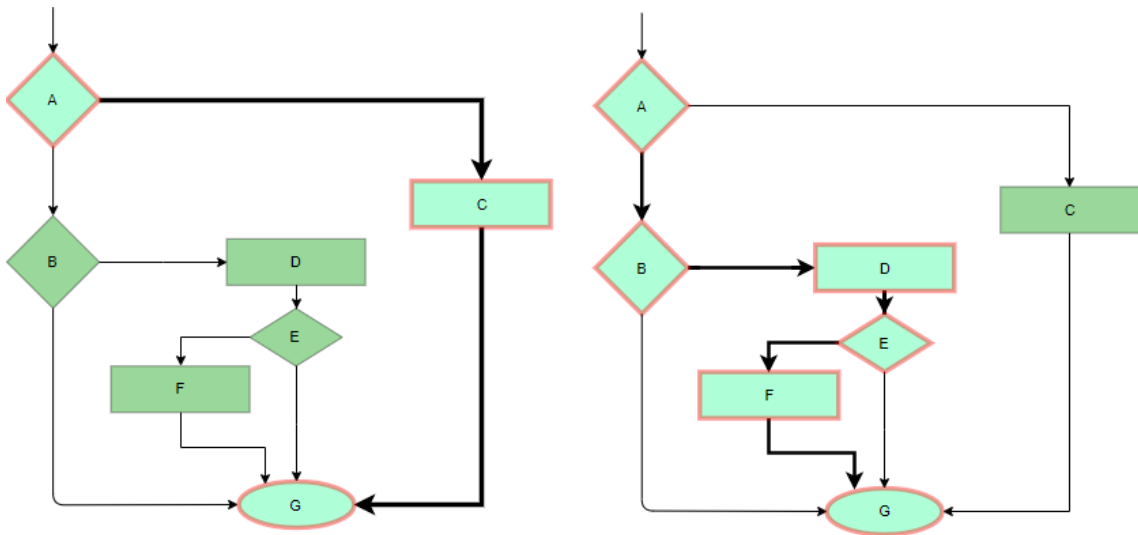
Statement Coverage Example

- **Branch Coverge:** In this technique, test cases are designed so that each branch from all decision points are traversed at least once. In a flowchart, all edges must be traversed at least once.
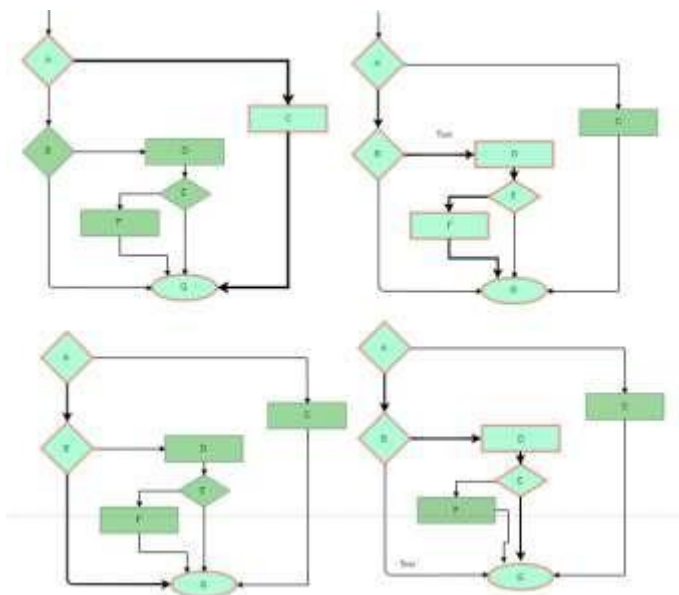
4 test cases required such that all branches of all decisions are covered, i.e, all edges of flowchart are covered

- **Condition Coverage:** In this technique, all individual conditions must be covered as shown in the following example:

-

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

  In this example, there are 2 conditions: X == 0 and Y == 0. Now, test these conditions get TRUE and FALSE as their values. One possible example would be:

  2. #TC1 – X = 0, Y = 55
  3. #TC2 – X = 5, Y = 0

- **Multiple Condition Coverage:** In this technique, all the possible combinations of the possible outcomes of conditions are tested at least once. Let's consider the following example:

  0. READ X, Y
  1. IF(X == 0 || Y == 0)
  2. PRINT '0'
  1. #TC1: X = 0, Y = 0
  2. #TC2: X = 0, Y = 5
  3. #TC3: X = 55, Y = 0
  4. #TC4: X = 55, Y = 5

  Hence, four test cases required for two individual conditions. Similarly, if there are n conditions then 2^n test cases would be required.

- **Basis Path Testing:** In this technique, control flow graphs are made from code or flowchart and then Cyclomatic complexity is calculated which defines the number of independent paths so that the minimal number of test cases can be designed for each independent path.

  Steps:
  0. Make the corresponding control flow graph
  1. Calculate the cyclomatic complexity
  2. Find the independent paths
  3. Design test cases corresponding to each independent path

Flow graph notation: It is a directed graph consisting of nodes and edges. Each node represents a sequence of statements, or a decision point. A predicate node is the one that represents a decision point that contains a condition after which the graph splits. Regions are bounded by nodes and edges.


if-else

until

while
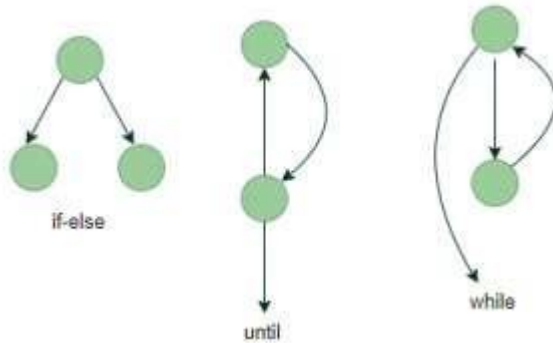
Cyclomatic Complexity: It is a measure of the logical complexity of the software and is used to define the number of independent paths. For a graph G, V(G) is its cyclomatic complexity.

Calculating V(G):

4. $V(G) = P + 1$, where P is the number of predicate nodes in the flow graph
5. $V(G) = E - N + 2$, where E is the number of edges and N is the total numberof nodes
6. $V(G) =$ Number of non-overlapping regions in the graph

Example:
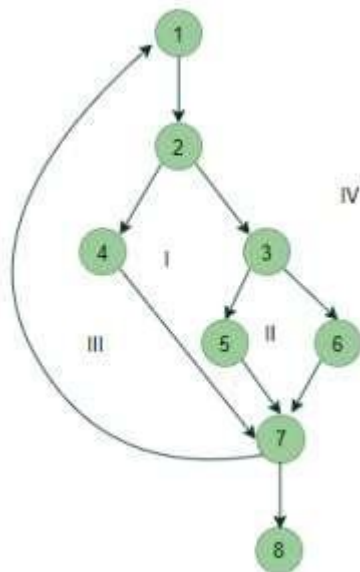
V(G)   =   4   (Using   any   of   the   above   formulae)

No of independent paths = 4

1. #P1: 1 – 2 – 4 – 7 – 8
2. #P2: 1 – 2 – 3 – 5 – 7 – 8
3. #P3: 1 – 2 – 3 – 6 – 7 – 8
4. #P4: 1 – 2 – 4 – 7 – 1 – . . . – 7 – 8

- **Loop Testing:** Loops are widely used and these are fundamental to many algorithms hence, their testing is very important. Errors often occur at the beginnings and ends of loops.

  0. **Simple loops:** For simple loops of size n, test cases are designed that:
      - Skip the loop entirely
      - Only one pass through the loop
      - 2 passes
      - m passes, where m < n
      - n-1 ans n+1 passes

  1. **Nested loops:** For nested loops, all the loops are set to their minimum count and we start from the innermost loop. Simple loop tests are conducted for the innermost loop and this is worked outwards till all the loops have been tested.

  2. **Concatenated loops:** Independent loops, one after another. Simple loop tests are applied for each.

      If they're not independent, treat them like nesting.

Advantages:

1. White box testing is very thorough as the entire code and structures are tested.
2. It results in the optimization of code removing error and helps in removing extra linesof code.
3. It can start at an earlier stage as it doesn't require any interface as in case of black box testing.
4. Easy to automate.

Disadvantages:

1. Main disadvantage is that it is very expensive.
2. Redesign of code and rewriting code needs test cases to be written again.
3. Testers are required to have in-depth knowledge of the code and programming language as opposed to black box testing.
4. Missing functionalities cannot be detected as the code that exists is tested.
5. Very complex and at times not realistic.

UNIT-7

Software Quality and Maintenance

Software Maintenance

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

Need for Maintenance –

Software Maintenance must be performed in order to:

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

Categories of Software Maintenance –

Maintenance can be divided into the following:

1. **Corrective maintenance:**

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

2. **Adaptive maintenance:**

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

3. **Perfective maintenance:**

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

4. **Preventive maintenance:**

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

Reverse Engineering –

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

Software Reverse Engineering –

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

<center>**Software Quality Assurance**</center>

What is Quality?

Quality defines to any measurable characteristics such as correctness, maintainability, portability, testability, usability, reliability, efficiency, integrity, reusability, and interoperability.

There are two kinds of Quality:

Quality of Design: Quality of Design refers to the characteristics that designers specify for an item. The grade of materials, tolerances, and performance specifications that all contribute to the quality of design.

Quality of conformance: Quality of conformance is the degree to which the design specifications are followed during manufacturing. Greater the degree of conformance, the higher is the level of quality of conformance.

Software Quality: Software Quality is defined as the conformance to explicitly state functional and performance requirements, explicitly documented development standards, and inherent characteristics that are expected of all professionally developed software.

Quality Control: Quality Control involves a series of inspections, reviews, and tests used throughout the software process to ensure each work product meets the requirements place upon it. Quality control includes a feedback loop to the process that created the work product.

Quality Assurance: Quality Assurance is the preventive set of activities that provide greater confidence that the project will be completed successfully.

Quality Assurance focuses on how the engineering and management activity will be done?

As anyone is interested in the quality of the final product, it should be assured that we are building the right product.

It can be assured only when we do inspection & review of intermediate products, if there are any bugs, then it is debugged. This quality can be enhanced.

Importance of Quality

We would expect the quality to be a concern of all producers of goods and services. However, the distinctive characteristics of software and in particular its intangibility and complexity, make special demands.

Increasing criticality of software: The final customer or user is naturally concerned about the general quality of software, especially its reliability. This is increasing in the case as organizations become more dependent on their computer systems and software is used more and more in safety-critical areas. For example, to control aircraft.

The intangibility of software: This makes it challenging to know that a particular task in a project has been completed satisfactorily. The results of these tasks can be made tangible by demanding that the developers produce 'deliverables' that can be examined for quality.

Accumulating errors during software development: As computer system development is made up of several steps where the output from one level is input to the next, the errors in the earlier ?deliverables? will be added to those in the later stages leading to accumulated determinable effects. In general the later in a project that an error is found, the more expensive it will be to fix. In addition, because the number of errors in the system is unknown, the debugging phases of a project are particularly challenging to control.

Software Quality Assurance Software quality assurance is a planned and systematic plan of all actions necessary to provide adequate confidence that an item or product conforms to establish technical requirements.

A set of activities designed to calculate the process by which the products are developed or manufactured.

Capability Maturity Model

Software Engineering Institute (SEI) Capability Maturity Model (CMM) specifies an increasing series of levels of a software development organization. The higher the level, the better the software development process, hence reaching each level is an expensive and time-consuming process.

Levels of CMM
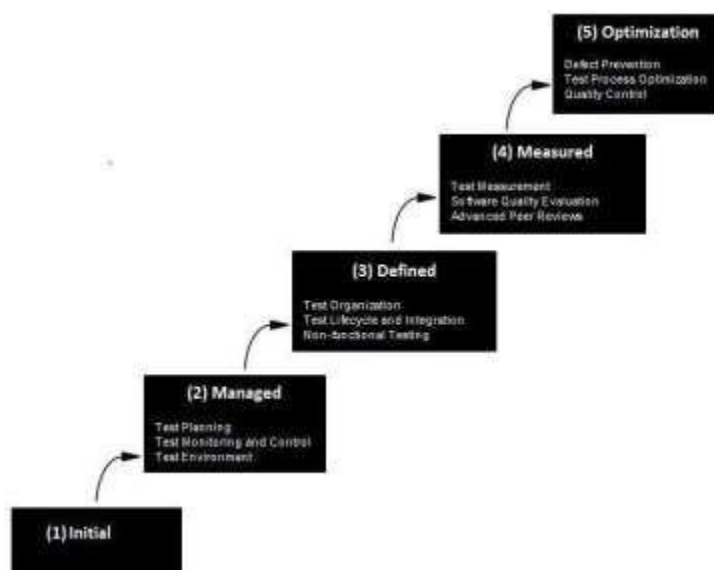
- **Level One :Initial** - The software process is characterized as inconsistent, and occasionally even chaotic. Defined processes and standard practices that exist are abandoned during a crisis. Success of the organization majorly depends on an individual effort, talent, and heroics. The heroes eventually move on to other organizations taking their wealth of knowledge or lessons learnt with them.

- **Level Two: Repeatable** - This level of Software Development Organization has a basic and consistent project management processes to track cost, schedule, and functionality. The process is in place to repeat the earlier successes on projects with similar applications. Program management is a key characteristic of a level two organization.

- **Level Three: Defined** - The software process for both management and engineering activities are documented, standardized, and integrated into a standard software process for the entire organization and all projects across the organization use an approved, tailored version of the organization's standard software process for developing,testing and maintaining the application.

- **Level Four: Managed** - Management can effectively control the software development effort using precise measurements. At this level, organization set a quantitative quality goal for both software process and software maintenance. At this maturity level, the performance of processes is controlled using statistical and other quantitative techniques, and is quantitatively predictable.

- **Level Five: Optimizing** - The Key characteristic of this level is focusing on continually improving process performance through both incremental and innovative technological improvements. At this level, changes to the process are to improve the process performance and at the same time maintaining statistical probability to achieve the established quantitative process-improvement objectives.

ISO 9000

- ISO 9000 is defined as a set of international standards on quality management and quality assurance developed to help companies effectively document the quality system elements needed to maintain an efficient quality system. They are not specific to any one industry and can be applied to organizations of any size.

- ISO 9000 can help a company satisfy its customers, meet regulatory requirements, and achieve continual improvement. It should be considered to be a first step or the base level of a quality system.

ISO 9000:2015 principles of Quality Management

The ISO 9000:2015 and ISO 9001:2015 standards are based on seven quality management principles that senior management can apply to promote organizational improvement.

ISO 9000 Quality Management Principles

1. Customer focus
   o Understand the needs of existing and future customers
   o Align organizational objectives with customer needs and expectations
   o Meet customer requirements
   o Measure customer satisfaction
   o Manage customer relationships
   o Aim to exceed customer expectations
   o Learn more about the customer experience and customer satisfaction

2. Leadership
   o Establish a vision and direction for the organization
   o Set challenging goals
   o Model organizational values
   o Establish trust
   o Equip and empower employees
   o Recognize employee contributions
   o Learn more about leadership

3. Engagement of people
   o Ensure that people's abilities are used and valued

- o Make people accountable
- o Enable participation in continual improvement
- o Evaluate individual performance
- o Enable learning and knowledge sharing
- o Enable open discussion of problems and constraints
- o Learn more about employee involvement

4. Process approach
- o Manage activities as processes
- o Measure the capability of activities
- o Identify linkages between activities
- o Prioritize improvement opportunities
- o Deploy resources effectively
- o Learn more about a process view of work and see process analysis tools

5. Improvement
- o Improve organizational performance and capabilities
- o Align improvement activities
- o Empower people to make improvements
- o Measure improvement consistently
- o Celebrate improvements
- o Learn more about approaches to continual improvement

6. Evidence-based decision making
- o Ensure the accessibility of accurate and reliable data
- o Use appropriate methods to analyze data
- o Make decisions based on analysis
- o Balance data analysis with practical experience
- o See tools for decision making

7. Relationship management
- o Identify and select suppliers to manage costs, optimize resources, and create value
- o Establish relationships considering both the short and long term
- o Share expertise, resources, information, and plans with partners
- o Collaborate on improvement and development activities
- o Recognize supplier successes

<h1>Six Sigma</h1>

Digital transformation has become the hottest buzzword of this decade. New technologies and tools are supporting the transformation journey of companies big and small as they compete to get a bigger slice of business in a fast-paced competitive environment. Yet, is it enough to smooth a company's transformative process? Can a standalone technology implementation remove a bottleneck in the production process or support troubleshooting a service design flaw? Although digital transformation fast-tracks a company's growth, it has to be equally supported by management methods of quality control and business transformation.

Keeping in tune with emerging markets and processes, the American company Motorola developed a new concept of quality management process in 1986. Over the years, it has been refined and polished into a sound theory of principles and methods, aimed at business transformation through a clearly defined process. This finished product is Six Sigma.

What is Six Sigma?

Six Sigma is a set of management tools and techniques designed to improve business by reducing the likelihood of error. It is a data-driven approach that uses a statistical methodology for eliminating defects.

The etymology is based on the Greek symbol "sigma" or "σ," a statistical term for measuring process deviation from the process mean or target. "Six Sigma" comes from the bell curve used in statistics, where one Sigma symbolizes a single standard deviation from the mean. If the process has six Sigmas, three above and three below the mean, the defect rate is classified as "extremely low."

The graph of the normal distribution below underscores the statistical assumptions of the Six Sigma model. The higher the standard deviation, the higher is the spread of values
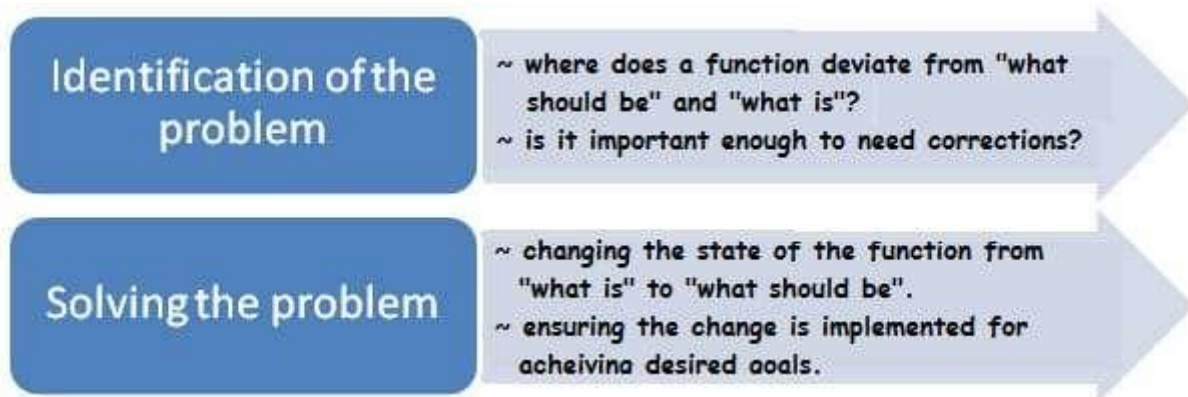
encountered. So, processes, where the mean is minimum 6σ away from the closest specification limit, are aimed at Six Sigma.
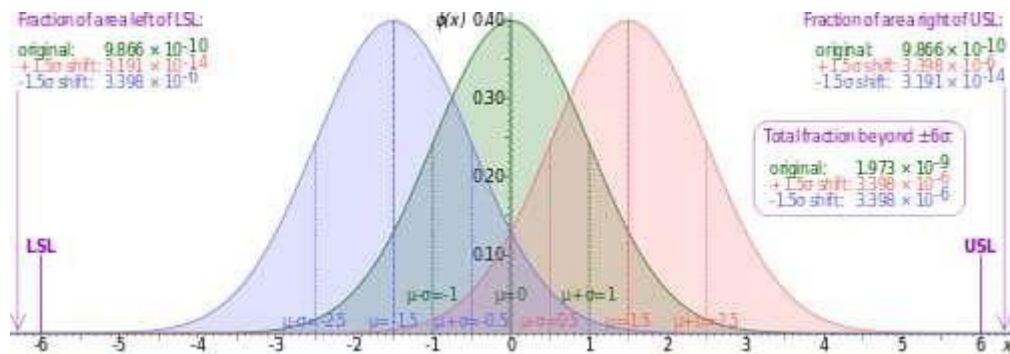
The 5 Key Principles of Six Sigma

The concept of Six Sigma has a simple goal – delivering near-perfect goods and services for business transformation for optimal customer satisfaction (CX).

Goals are achieved through a two-pronged approach:

Six Sigma has its foundations in five key principles:

1.   **Focus on the Customer**

This is based on the popular belief that the "customer is the king." The primary goal is to bring maximum benefit to the customer. For this, a business needs to understand its customers, their needs, and what drives sales or loyalty. This requires establishing the standard of quality as defined by what the customer or market demands.

2.   Measure the Value Stream and Find Your Problem

Map the steps in a given process to determine areas of waste. Gather data to discover the specific problem area that is to be addressed or transformed. Have clearly defined goals for data collection, including defining the data to be collected, the reason for the data gathering, insights expected, ensuring the accuracy of measurements, and establishing a standardized data collection system. Ascertain if the data is helping to achieve the goals, whether or not the data needs to be refined, or additional information collected. Identify the problem. Ask questions and find the root cause.

3.   Get Rid of the Junk

Once the problem is identified, make changes to the process to eliminate variation, thus removing defects. Remove the activities in the process that do not add to the customer value. If the value stream doesn't reveal where the problem lies, tools are used to help discover the outliers and problem areas. Streamline functions to achieve quality control and efficiency. In the end, by taking out the above-mentioned junk, bottlenecks in the process are removed.

4.   Keep the Ball Rolling

Involve all stakeholders. Adopt a structured process where your team contributes and collaborates their varied expertise for problem-solving.

Six Sigma processes can have a great impact on an organization, so the team has to be proficient in the principles and methodologies used. Hence, specialized training and knowledge are required to reduce the risk of project or re-design failures and ensure that the process performs optimally.

5.   Ensure a Flexible and Responsive Ecosystem

The essence of Six Sigma is business transformation and change. When a faulty or inefficient process is removed, it calls for a change in the work practice and employee approach. A robust culture of flexibility and responsiveness to changes in procedures can ensure streamlined project implementation. The people and departments involved should be able to adapt to change with ease, so to facilitate this, processes should be designed for quick and seamless adoption. Ultimately, the company that has an eye fixed on the data examines the bottom line periodically and adjusts its processes where necessary, can gain a competitive edge.

The Six Sigma Methodology

The two main Six Sigma methodologies are DMAIC and DMADV. Each has its own set of recommended procedures to be implemented for business transformation.

DMAIC is a data-driven method used to improve existing products or services for better customer satisfaction. It is the acronym for the five phases: D – Define, M – Measure, A – Analyse, I – Improve, C – Control. DMAIC is applied in the manufacturing of a product or delivery of a service.

DMADV is a part of the Design for Six Sigma (DFSS) process used to design or re-design different processes of product manufacturing or service delivery. The five phases of DMADV are: D – Define, M – Measure, A – Analyse, D – Design, V – Validate. DMADV is employed when existing processes do not meet customer conditions, even after optimization, or when it is required to develop new methods. It is executed by Six Sigma Green Belts and Six Sigma Black Belts and under the supervision of Six Sigma Master Black Belts.
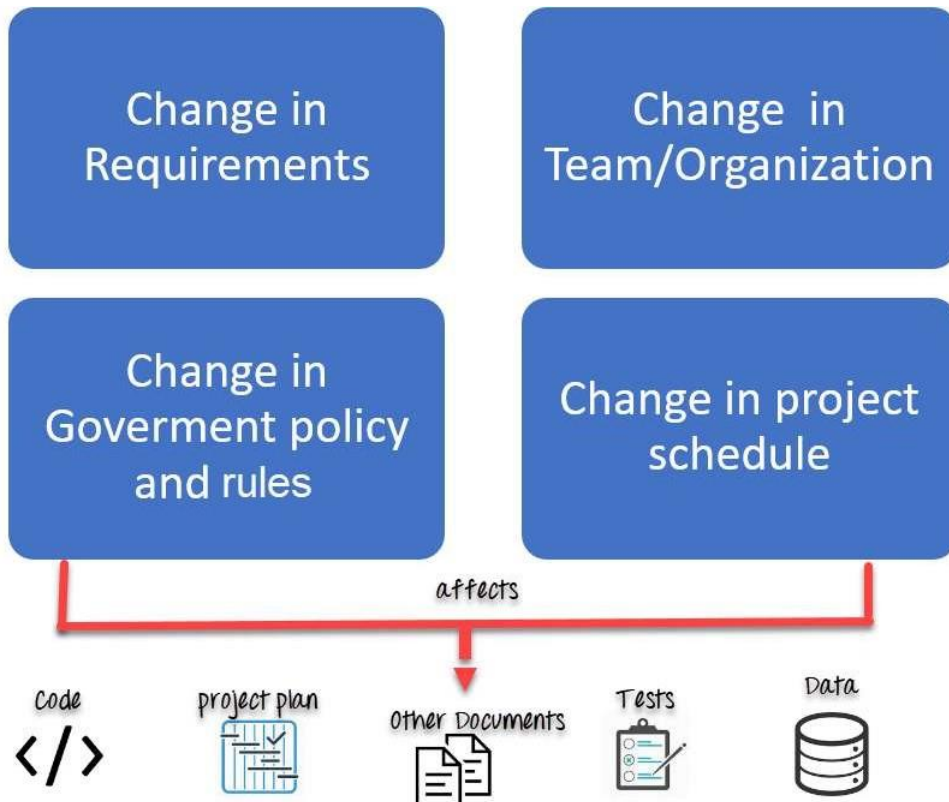
Software Configuration Management(SCM)

Software Configuration Management(SCM) is a process to systematically manage, organize, and control the changes in the documents, codes, and other entities during the

Software Development Life Cycle. The primary goal is to increase productivity with minimal mistakes. SCM is part of cross-disciplinary field of configuration management and it can accurately determine who made which revision.

Why do we need Configuration management?

The primary reasons for Implementing Technical Software Configuration Management System are:

- There are multiple people working on software which is continually updating
- It may be a case where multiple version, branches, authors are involved in a software config project, and the team is geographically distributed and works concurrently
- Changes in user requirement, policy, budget, schedule need to be accommodated.
- Software should able to run on various machines and Operating Systems
- Helps to develop coordination among stakeholders
- SCM process is also beneficial to control the costs involved in making changes to a system

Any change in the software configuration Items will affect the final product. Therefore, changes to configuration items need to be controlled and managed.

Tasks in SCM process

- Configuration Identification
- Baselines
- Change Control
- Configuration Status Accounting
- Configuration Audits and Reviews

Configuration Identification:

Configuration identification is a method of determining the scope of the software system. With the help of this step, you can manage or control something even if you don't know what it is. It is a description that contains the CSCI type (Computer Software Configuration Item), a project identifier and version information.

Activities during this process:

- Identification of configuration Items like source code modules, test case, and requirements specification.
- Identification of each CSCI in the SCM repository, by using an object-oriented approach
- The process starts with basic objects which are grouped into aggregate objects. Details of what, why, when and by whom changes in the test are made
- Every object has its own features that identify its name that is explicit to all other objects
- List of resources required such as the document, the file, tools, etc.

Example:

Instead of naming a File login.php its should be named login_v1.2.php where v1.2 stands for the version number of the file

Instead of naming folder "Code" it should be named "Code_D" where D represents code should be backed up daily.

Baseline:

A baseline is a formally accepted version of a software configuration item. It is designated and fixed at a specific time while conducting the SCM process. It can only be changed through formal change control procedures.

Activities during this process:

- Facilitate construction of various versions of an application
- Defining and determining mechanisms for managing various versions of these work products
- The functional baseline corresponds to the reviewed system requirements
- Widely used baselines include functional, developmental, and product baselines

In simple words, baseline means ready for release.

Change Control:

Change control is a procedural method which ensures quality and consistency when changes are made in the configuration object. In this step, the change request is submitted to software configuration manager.

Activities during this process:

- Control ad-hoc change to build stable software development environment. Changes are committed to the repository
- The request will be checked based on the technical merit, possible side effects and overall impact on other configuration objects.
- It manages changes and making configuration items available during the software lifecycle

Configuration Status Accounting:

Configuration status accounting tracks each release during the SCM process. This stage involves tracking what each version has and the changes that lead to this version.

Activities during this process:

- Keeps a record of all the changes made to the previous baseline to reach a new baseline
- Identify all items to define the software configuration
- Monitor status of change requests
- Complete listing of all changes since the last baseline
- Allows tracking of progress to next baseline
- Allows to check previous releases/versions to be extracted for testing

Configuration Audits and Reviews:

Software Configuration audits verify that all the software product satisfies the baseline needs. It ensures that what is built is what is delivered.

Activities during this process:

- Configuration auditing is conducted by auditors by checking that defined processes are being followed and ensuring that the SCM goals are satisfied.
- To verify compliance with configuration control standards. auditing and reporting the changes made
- SCM audits also ensure that traceability is maintained during the process.
- Ensures that changes made to a baseline comply with the configuration status reports
- Validation of completeness and consistency