Basic Concepts in Software Engineering and Software Project Management
Unit – I

Abstraction versus Decomposition

In software engineering there are two main basic concepts abstraction and decomposition. Both concepts are analysis & design basic techniques. There are interrelated and normally used together during software development.

Abstraction

Abstraction in general is process of consciously ignoring some aspects of a subject under analysis in order to better understand other aspects of it. In other words, it kind of simplification of a subject. In software in particular, analysis & design are all about abstraction. Abstraction is one of the fundamental principles of object-oriented programming. Abstraction allows us to name objects that are not directly instantiated but serve as a basis for creating objects with some common attributes or properties. For example: in the context of computer accessories Data Storage Device is an abstract term because it can either be a USB pen drive, hard disk, or RAM. But a USB pen drive or hard disks are concrete objects because their attributes and behaviors are easily identifiable, which is not the case for Data Storage Device, being an abstract object for computer accessories. So, abstraction is used to generalize objects into one category in the design phase. For example, in a travel management system you can use Vehicle as an abstract object or entity that generalizes how you travel from one place to another.

Decomposition

Decomposition is an application of the old good principle "divide and conquer" to software development. It is a technique of classifying, structuring and grouping complex elements in order to end up with more atomic ones, organized in certain fashion and easier to manage. Decomposition is a way to break down your systems into modules in such a way that each module provides different functionality, but may affect other modules also. To understand decomposition quite clearly, you should first understand the concepts of association, composition, and aggregation.

Evolution of Software Engineering Techniques

Software evolution is a term which refers to the process of developing software initially,

then timely updating it for various reasons, i.e., to add new features or to remove obsolete functionalities etc. The evolution process includes fundamental activities of change analysis, release planning, system implementation and releasing a system to customers. The cost and impact of these changes are accessed to see how much system is affected by the change and how much it might cost to implement the change. If the proposed changes are accepted, a new release of the software system is planned. During release planning, all the proposed changes (fault repair, adaptation, and new functionality) are considered. A design is then made on which changes to implement in the next version of the system. The process of change implementation is an iteration of the development process where the revisions to the system are designed, implemented and tested.

The Necessity of Software Evolution

Software evaluation is necessary just because of the following reasons

a) Change in requirement with time: With the passes of time, the organization's needs and modus Operandi of working could substantially be changed so in this frequently changing time the tools (software) that they are using need to change for maximizing the performance.

b) Environment change: As the working environment changes the things(tools) that enable us to work in that environment also changes proportionally same happens in the software world as the working environment changes then, the organizations need reintroduction of old software with updated features and functionality to adapt the new environment.

c) Errors and bugs: As the age of the deployed software within an organization increases their preciseness or impeccability decrease and the efficiency to bear the increasing complexity workload also continually degrades. So, in that case, it becomes necessary to avoid use of obsolete and aged software. All such obsolete Software's need to undergo the evolution process in order to become robust as per the workload complexity of the current environment.

d) Security risks: Using outdated software within an organization may lead you to at the verge of various software-based cyber-attacks and could expose your confidential data illegally associated with the software that is in use. So, it becomes necessary to avoid such security breaches through regular assessment of the security patches/modules are used within the software. If the software isn't robust enough to bear the current

occurring Cyber-attacks so it must be changed (updated).

e) New functionality and features: In order to increase the performance and fast data processing and other functionalities, an organization need to continuously evolutes the software throughout its life cycle so that stakeholders & clients of the product could work efficiently.

Laws used for Software Evolution

Law of continuing change: This law states that any software system that represents some real-world reality undergoes continuous change or become progressively less useful in that environment.

Law of increasing complexity: As an evolving program changes, its structure becomes more complex unless effective efforts are made to avoid this phenomenon.

Law of conservation of organization stability: Over the lifetime of a program, the rate of development of that program is approximately constant and independent of the resource devoted to system development.

Law of conservation of familiarity: This law states that during the active lifetime of the program, changes made in the successive release are almost constant.

Software Development Life Cycle (SDLC) Models

Software development life cycle (SDLC) is a series of phases that provide a common understanding of the software building process. How the software will be realized and developed from the business understanding and requirements elicitation phase to convert these business ideas and requirements into functions and features until its usage and operation to achieve the business needs. The good software engineer should have enough knowledge on how to choose the SDLC model based on the project context and the business requirements.

The Software Process

A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another. A process framework establishes the foundation for a complete software engineering process by identifying a small number of framework

activities that are applicable to all software projects, regardless of their size or complexity. The process framework encompasses a set of umbrella activities that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities.

Framework Activities

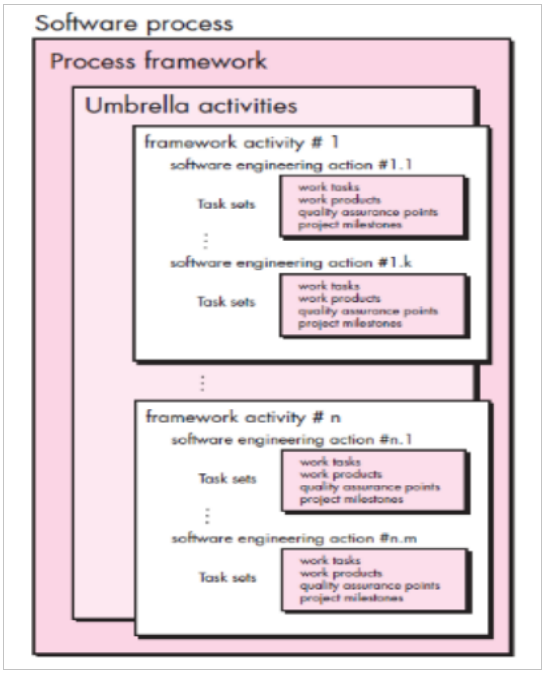
Communication: Before any technical work can commence, it is critically important to communicate and collaborate with the customer (end user). Objectives for the project and to gather requirements that help define software features and functions.

Planning: The planning activity creates a “map” that helps guide the team as it makes the journey. The map called a software project plan - defines the software engineering work by describing the technical tasks to be conducted.

Modeling: A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

Construction: This activity combines code generation and the testing that is required uncovering errors in the code.

Deployment: The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.



Umbrella Activities

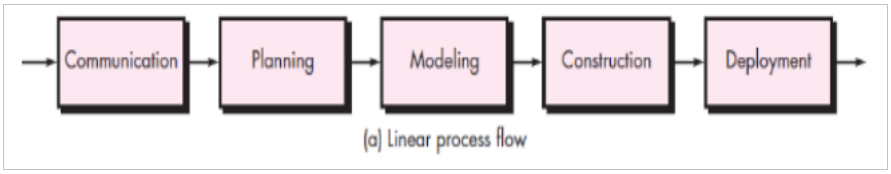
- Software Project Tracking and Control
- Risk Management
- Software Quality Assurance
- Technical Reviews
- Measurement
- Software Configuration Management
- Reusability Management
- Work Product Preparation and Production

Note: Software Process=Framework Activities + Umbrella Activities

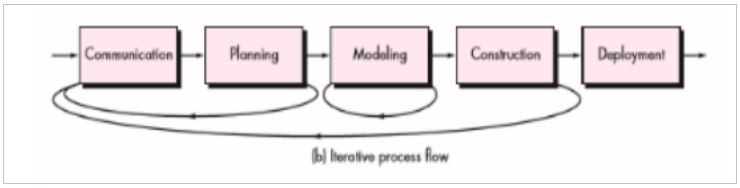
A Generic Process Models

Process model is a standard format for planning, organizing and running a development project. It helps in the software development. These models are guiding the software development team. Any process model includes set of frame work activities begin with communication ends with deployment. These are models are

1. **A linear process flow** executes each of the five framework activities in sequence, beginning with communication and culminating with deployment.

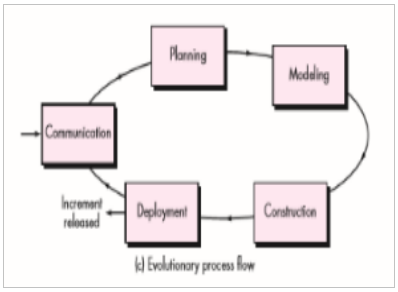


2. **An iterative process flow** repeats one or more of the activities before proceeding to the next.

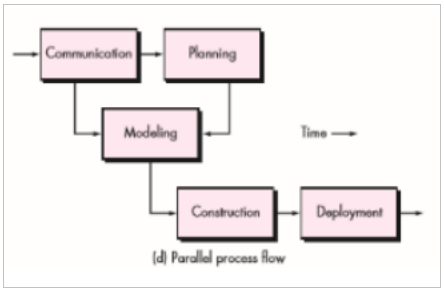


3. **An evolutionary process flow** executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the

software.

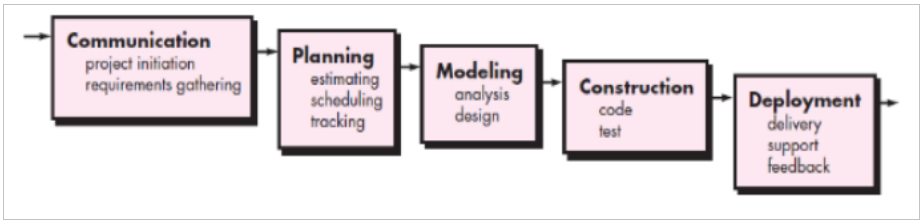


4. **A parallel process flow** executes one or more activities in parallel with other activities.



1. Waterfall Model

The waterfall model is the oldest paradigm for software engineering. It is also called as the classic life cycle, suggests a systematic, sequential approach6 to software development that begins with customer specification of requirements and progresses through planning, modeling, construction and deployment. This model is oldest and widely used model followed in software engineering.



Phases of waterfall model

- ✔ Communication
- ✔ Planning
- ✔ Modeling
- ✔ Construction
- ✔ Deployment

Advantages

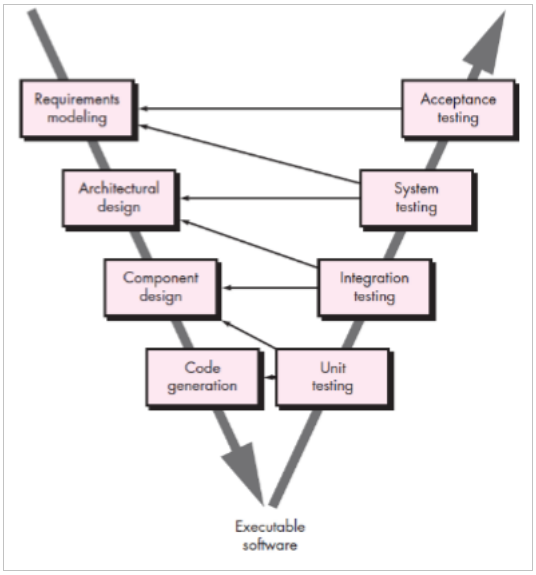
- Simple and easy to understand and use.
- It allows for departmentalization and control.
- Works well for small projects because requirements are very well understand.
- A schedule can be set with deadlines for each stage of development.
- Clearly defined stages.
- Process and results are well document.

Disadvantages

- Late process.
- High amounts of risk and uncertainty.
- Not good model for complex, long and ongoing projects.
- Integration is done as a big bang at very end.

2. V-Model

A variation in the representation of the waterfall model is called the V-model. In this Verification and Validation actions are associated with earlier engineering actions. The V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities. As software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution. Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.



3. Iterative Waterfall Model

In a practical software development project, the classical waterfall model is hard to use. So, the Iterative waterfall model can be thought of as incorporating the necessary changes to the classical waterfall model to make it usable in practical software development projects. It is almost the same as the classical waterfall model except some changes are made to increase the efficiency of the software development. The iterative waterfall model provides feedback paths from every phase to its preceding phases, which is the main difference from the classical waterfall model.

Iterative Waterfall Model is the extension of the Waterfall model. This model is almost same as the waterfall model except some modifications are made to improve the performance of the software development. The iterative waterfall model provides customer's feedback paths from each phase to its previous phases. There is no feedback path provided for feasibility study phase, so if any change is required in that phase then iterative model doesn't have scope for modification or making corrections.

Iterative waterfall allows going back on the previous phase and change the requirements and some modification can done if necessary. This model reduces the developer's effort and time required to detect and correct the errors. In iterative

waterfall model, next phase can only begins when the previous phase is completed as waterfall model.

Phases of Iterative Waterfall Model

- ✔ Requirement Analysis
- ✔ Feasibility Study
- ✔ Software Design
- ✔ Coding/Implementation
- ✔ Software Testing
- ✔ Software Deployment
- ✔ Software Maintenance
- ✔ Iterative Waterfall Model

Advantages

- Iterative waterfall model is very easy to understand and use.
- Every phase contains feedback path to its previous phase.
- This is an simple to make changes or any modifications at any phase.
- By using this model, developer can completer project earlier.
- Customer involvement is not required during the software development.
- This model is suitable for large and complex projects.

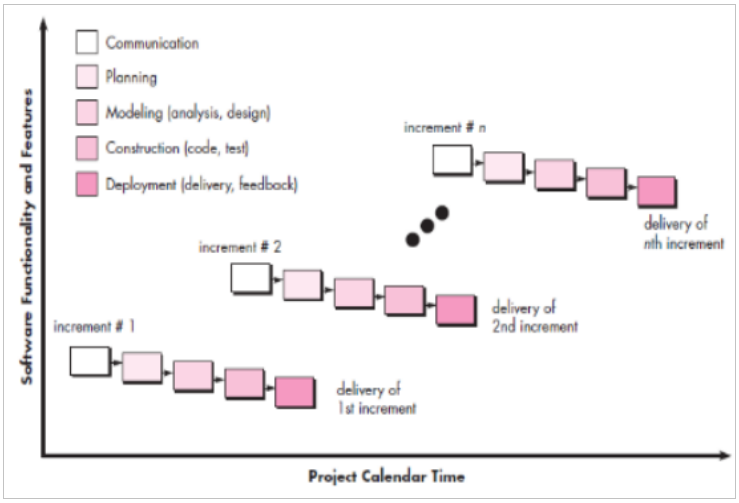
Disadvantages

- There is no feedback path for feasibility study phase.

- This model is not suitable if requirements are not clear.
- It can be more costly.
- There is no process for risk handling.
- Customer can view the final project. there is no prototype for taking customer reviews.
- This model does not work well for short projects.
- If modifications are required repeatedly then it can be more complex projects.

4. Incremental Model

The incremental model combines elements of waterfall model with iterative fashion. The incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable “increments” of the software in a manner that is similar to the increments produced by an evolutionary process flow.



Phases of Incremental model

- ✔ Communication
- ✔ Planning
- ✔ Modeling
- ✔ Construction
- ✔ Deployment

Advantages

- This model is more flexible and less costly to change scope and requirements.
- Easier to test and debug during a smaller iterations.
- Easier to manage risk.
- Customer can respond to each built.
- Low initial cost.

Disadvantages

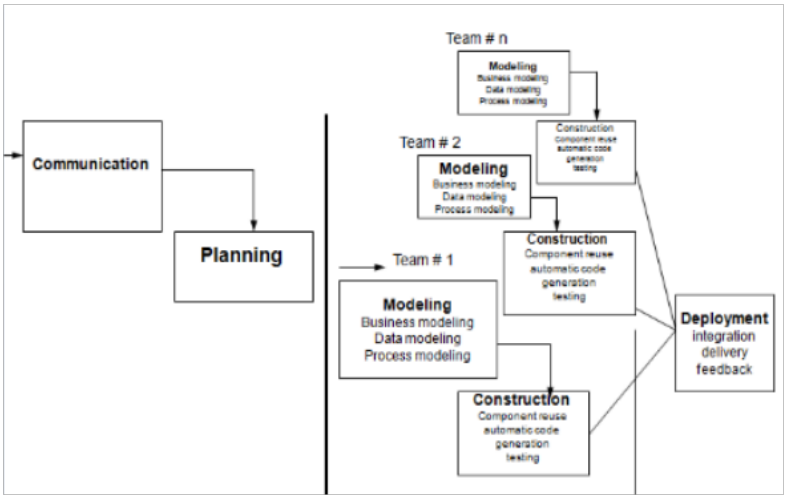
- Needs good planning and design.
- Needs a clear and complete definition of the whole system before development.
- Total cost is higher than waterfall model.

5. RAD Model

RAD stands for Rapid Application Development is an incremental software process model that focuses on short development cycle time. This model is high speed model which adapts many steps from waterfall model in which rapid development is achieved by using component based construction approach. If project requirements are well understood and project scope is well known then RAD process enables a development team to create a full functional system.

Phases of RAD model

- ✔ Communication
- ✔ Planning
- ✔ Modeling
    - o Business modeling
    - o Data modeling
    - o Process modeling
- ✔ Construction
- ✔ Deployment



Advantages

- Reduced development time.
- Increases reusability of components.
- Encourage customer feedback.
- Quick initial reviews.

Disadvantages

- High dependency on modeling skills.
- Unknown cost of product.
- Difficult for many important users to commit the time required for success of the RAD process.

6. Evolutionary Models

Evolutionary process models produce an increasingly more complete version of the software with each iteration. These models are the complex systems, evolves over a period of time. Evolutionary models are iterative and evolutionary process flow models. These models are

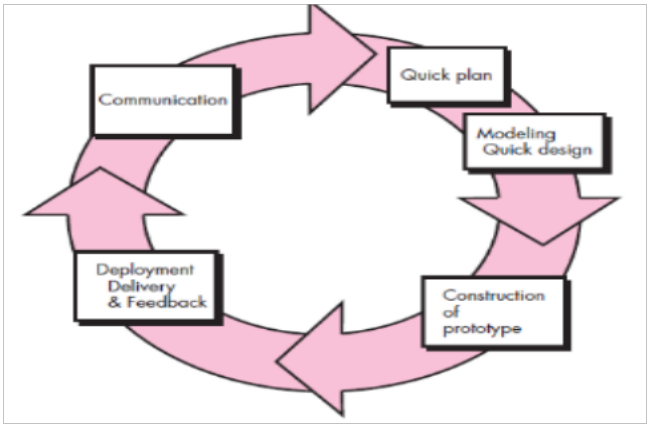
- ➤ Prototype model
- ➤ Spiral model

Phases of Evolutionary model

- ✔ Communication
- ✔ Planning
- ✔ Modeling
- ✔ Construction
- ✔ Deployment

Prototype Model

A customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human- machine interaction should take. In these, and many other situations, a prototyping paradigm may offer the best approach. Prototyping can be used as a stand-alone process model; it is more commonly used as a technique that can be implemented within the context of any one of the process models.



Advantages

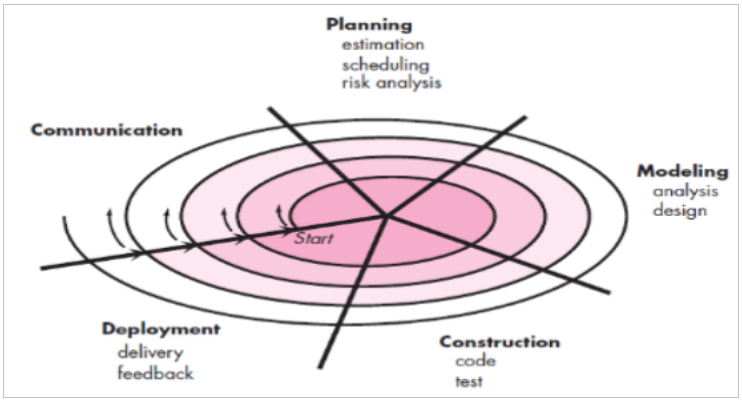
- A quick design occur
- Quick design leads to the construction of prototype
- Prototype is evaluated by the customer.
- Requirements are refined.
- Turned to satisfy the needs of customer.

Disadvantages

- Customer requirements do not satisfy the developer.
- It is slow process, not delivers in time.
- Too much involvement of client.
- Too many changes.

Spiral Model

The spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software. The spiral development model is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions. Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. A spiral model is divided into a set of framework activities defined by the software engineering team.



Advantages

- Changing requirements can be accommodated.
- Allows extensive use of prototypes.
- Requirements can be captured more accurately.
- Users see the system early.
- Helps in risk management (divided into small parts and the risky parts can be developed earlier).

Disadvantages

- Management is more complex.
- End of the project may not be known early.
- Process is complex.
- Large no of intermediate stages requires excessive documentation.
- Spiral may go on indefinitely.
- Not suitable for small or low risk projects.

Agile Models

Software engineering is littered with dozens of obsolete process descriptions and methodologies, modeling methods and notations, tools and technology. With the introduction of a wide array of agile process models each contending for acceptance within the software development community the agile movement is following the same historical path. Other agile process models have been proposed and are in use across the industry.

- Extreme Programming (XP)
- Adaptive Software Development (ASD)
- Dynamic Systems Development Method (DSDM)
- Scrum
- Crystal
- Feature Drive Development (FDD)
- Agile Modeling (AM)

Extreme Programming (XP)

Extreme programming (XP) is the most widely used agile process. Organized as four framework activities planning, design, coding, and testing XP suggests a number of innovative and powerful techniques that allow an agile team to create frequent software releases that deliver features and functionality that have been described and then prioritized by stakeholders.

Adaptive Software Development (ASD)

Adaptive Software Development (ASD) as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization. ASD uses an iterative process that incorporates adaptive cycle planning, relatively rigorous requirement gathering methods, and an iterative

development cycle that incorporates customer focus groups and formal technical reviews as real-time feedback mechanisms. An ASD “life cycle” that incorporates three phases: **speculation, collaboration and learning**.

Dynamic Systems Development Method (DSDM)

The Dynamic Systems Development Method (DSDM) is an agile software development approach that “provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment”.

DSDM is an iterative software process in which each iteration follows the 80 percent rule. DSDM is a process framework that can adopt the tactics of another agile approach such as XP.

Functional model iteration, Design and build iteration and Implementation

Scrum

Scrum (an activity) is an agile software development method. Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: **requirements, analysis, design, evolution, and delivery**. Within each framework activity, work tasks occur within a process pattern called a sprint.

Crystal

Crystal is a family of agile process models that can be adopted to the specific characteristics of a project. The Crystal family is actually a set of example agile processes that have been proven effective for different types of projects. The intent is to allow agile teams to select the member of the crystal family that is most appropriate for their project and environment.

Feature Drive Development (FDD)

Feature Driven Development (FDD) is somewhat more “formal” than other agile methods, but still maintains agility by focusing the project team on the development of features a client-valued function that can be implemented in two weeks or less. FDD adopts a philosophy that

1. Emphasizes collaboration among people on an FDD team.
2. Manages problem and project complexity using feature-based decomposition followed by the integration of software increments.

3. Communication of technical detail using verbal, graphical, and text-based means.

FDD emphasizes software quality assurance activities by encouraging an incremental development strategy, the use of design and code inspections, the application of software quality assurance audits, the collection of metrics, and the use of patterns. The emphasis on the definition of features provides the following benefits:

- Because features are small blocks of deliverable functionality, users can describe them more easily; understand how they relate to one another more readily; and better review them for ambiguity, error, or omissions.
- Features can be organized into a hierarchical business-related grouping.
- Since a feature is the FDD deliverable software increment, the team develops operational features every two weeks.
- Because features are small, their design and code representations are easier to inspect effectively.
- Project planning, scheduling, and tracking are driven by the feature hierarchy, rather than an arbitrarily adopted software engineering task set.

Agile Modeling (AM)

Agile modeling (AM) suggests that modeling is essential for all systems, but that the complexity, type and size of the model must be tuned to the software to be built. Agile modeling adopts all of the values that are consistent with the agile manifesto. The agile modeling philosophy recognizes that an agile team must have the courage to make decisions that may cause it to reject a design and refractor. The team must also have the humility to recognize that technologists do not have all the answers and that business expert and other stakeholders should be respected and embraced. Modeling principles are

Model with a purpose

Use multiple models

Travel light

Content is more important than representation

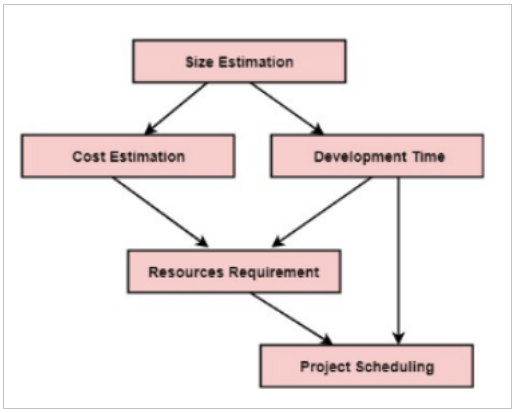
Know the models and the tools you use to create them

Adapt locally.

Project Planning

Software project planning is task, which is performed before the production of software

actually starts. Project planning is a process of continuous thinking of activities and requirements of doing project. A Software Project is the complete methodology of programming advancement from requirement gathering to testing and support, completed by the execution procedures, in a specified period to achieve intended software product. It is there for the software production but involves no concrete activity that has any direction connection with software production; rather it is a set of multiple processes, which facilitates software production. Software Project planning starts before technical work start. The various steps of planning activities are:



The size is the crucial parameter for the estimation of other activities. Resources requirement are required based on cost and development time. Project schedule may prove to be very useful for controlling and monitoring the progress of the project. This is dependent on resources & development time.

Project Estimation

Estimation is a process to predict the time and the cost that a project requires to be finished appropriately. Estimation is an attempt to determine how much money, efforts, resources and time it will take to build specific software-based system or project.

Who does estimation?

- Software manager does estimation using information collected from customers and software engineers
- Software metrics data collected from past projects.
- Assumption taken from experience .

- Identified risks and feasibility helps in estimation.

Steps for Estimations

- Estimate the size of the development product
- Estimate the effect in person- month or person – hours quality
- Estimate the schedule in calendar months.
- Estimate the project cost in agreed currency.

Before a final estimate is made, problem complexity and risk are considered. "Estimate Risk" is measured by the degree of uncertainty in the quantitative estimate established by the resources, and schedule.

Project Scheduling

A schedule in your project's time table actually consists of sequenced activities and milestones that are needed to be delivered under a given period of time. Project schedule simply means a mechanism that is used to communicate and know about that task are needed and has to be done or performed and which organizational resources will be given or allocated to these tasks and in what time duration or time frame work is needed to be performed. Effective project scheduling leads to success of project, reduced cost, and increased customer satisfaction. Scheduling in project management means to list out activities, deliverables, and milestones within a project that are delivered. It contains more notes than your average weekly planner notes. The most common and important form of project schedule is Gantt chart.

The manager needs to estimate time and resources of project while scheduling project. All activities in project must be arranged in a coherent sequence that means activities should be arranged in a logical and well-organized manner for easy to understand. Initial estimates of project can be made optimistically which means estimates can be made when all favorable things will happen and no threats or problems take place. The total work is separated or divided into various small activities or tasks during project schedule. Then, Project manager will decide time required for each activity or task to get completed. Even some activities are conducted and performed in parallel for efficient performance. The project manager should be aware of fact that each stage of project is not problem-free.

Problems arise during Project Development Stage

People may leave or remain absent during particular stage of development. Hardware may get failed while performing. Software resource that is required may not be available at present, etc. The project schedule is represented as set of charts in which work-breakdown structure and dependencies within various activities are represented. To accomplish and complete project within a given schedule, required resources must be available when they are needed. Therefore, resource estimation should be done before starting development.

Resources required for Development of Project

- Human effort
- Sufficient disk space on server
- Specialized hardware
- Software technology
- Travel allowance required by project staff, etc.

Advantages of Project Scheduling

There are several advantages provided by project schedule in our project management:

- It simply ensures that everyone remains on same page as far as tasks get completed, dependencies, and deadlines.
- It helps in identifying issues early and concerns such as lack or unavailability of resources.
- It also helps to identify relationships and to monitor process.
- It provides effective budget management and risk mitigation.

Halstead Software Science

Halstead complexity metrics were developed by the late Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands in the module to measure a program module's complexity directly from source code. Among the earliest software metrics, they are strong indicators of code complexity. Halstead Science is an estimation technique to find out size, time and effort of a software

Number of Operators and Operands

Halstead´s metrics is based on interpreting the source code as a sequence of tokens and classifying each token to be an operator or an operand.

First, we need to compute the following numbers, given the program:

n1 = the number of distinct operators

n2 = the number of distinct operands

N1 = the total number of operators

N2 = the total number of operands

The number of unique operators and operands (n1 and n2) as well as the total number of operators and operands (N1 and N2) are calculated by collecting the frequencies of each operator and operand token of the source program. From these numbers, following measures can be calculated:

Program length (N)

The program length (N) is the sum of the total number of operators and operands in the program:

$$N = N1 + N2$$

Vocabulary size (n)

The vocabulary size (n) is the sum of the number of unique operators and operands:

$$n = n1 + n2$$

Program volume (V)

The program volume (V) is the information contents of the program, measured in mathematical bits. It is calculated as the program length times the 2-base logarithm of the vocabulary size (n) :

V = N * log2(n)

Halstead's volume (V) describes the size of the implementation of an algorithm. The computation of V is based on the number of operations performed and operands handled in the algorithm. Therefore, V is less sensitive to code layout than the lines-of-code measures.

The volume of a function should be at least 20 and at most 1000. The volume of a parameter less one-line function that is not empty; is about 20. A volume greater than 1000 tells that the function probably does too many things. The volume of a file should be at least 100 and at most 8000. These limits are based on volumes measured for files whose LOCpro and v(G) are near their recommended limits. The limits of volume can be used for double-checking.

Difficulty level (D)

The difficulty level or error proneness (D) of the program is proportional to the number of unique operators in the program.

D is also proportional to the ration between the total number of operands and the number of unique operands (i.e., if the same operands are used many times in the program, it is more prone to errors).

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

Program level (L)

The program level (L) is the inverse of the error proneness of the program i.e., a low-level program is more prone to errors than a high-level program.

L = 1 / D

Effort to implement (E)

The effort to implement (E) or understand a program is proportional to the volume and to the difficulty level of the program.

E = V * D

Estimated Program Length

According to Halstead, the first hypothesis of software science is that the length of a well-Structured program is a function only of the number of unique operators and

operands, the estimated length is denoted by N^

N^= n1log2n1+n2log2n2

Potential Volume

Amongst all the programs, the one that has minimal size is said to have the potential volume, V*. Halstead argued that the minimal implementation of any algorithm was through a reference to a procedure that had been previously written. The implementation of this algorithm would then require nothing more than invoking the procedure and supplying the operands for its inputs and output parameters.

V*= (2+n2*) log2(2+n2*)

Staffing

Staffing management plan and resource management plans are important part of project resource management. Every project will require resources for executing project activities. There will be a need for both man power resources and physical resources. The resource requirement for each activity will be estimated. The resources will be acquired during project execution as per the schedule. Planning for resources, acquiring resources, developing team and managing team are the important activities to be carried out as part of project resource management. A resource management plan will contain all the necessary guidelines for project resource management. A staffing management plan will also be part of the overall resource management plan.

Staffing management plan, which part of overall resource management plan will specifically focus on the man power aspects of the project. Staffs are the most important part of project. It is important to select and acquire the right staff with right skills at the right time. A staffing management plan contains a plan for addressing all the aspects of man power and will include below information:

- Identification of human resources
- How the human resources will be acquired
- Criteria to be used for how the human resources will be selected
- From where the human resources will be acquired
- How to acquire resources from within the organization
- How to acquire resources from external sources
- When the resources will be acquired (based on the project schedule)
- When the resources will be released (based on the project schedule)

- Process for maintaining the resource calendars
- Resource loading table depicting total number of resources needed at different points in the project
- Safety and security guidelines for the human resources
- Identification of training needs and plan for fulfilling the training needs of the team
- Rewards and recognition plan for the team
- How to build the team and enhance team performance
- How to monitor the performance of each team member and help keeping them motivated

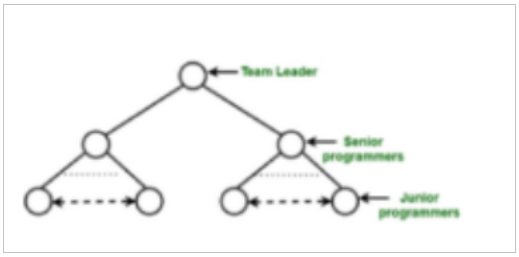
Organization and Team Structure

The project organizational structure is an essential configuration for determining the hierarchy of people, their function, workflow and reporting system. It is a factor in business that plays a fundamental role in guiding and defining the way in which the organization carries out its operations. There are many ways to organize the project team. Some important ways are as follows:

- Hierarchical Team Organization
- Chief-Programmer Team Organization
- Matrix Team Organization
- Egoless Team Organization
- Democratic Team Organization

Hierarchical Team Organization

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



Chief-Programmer Team Organization

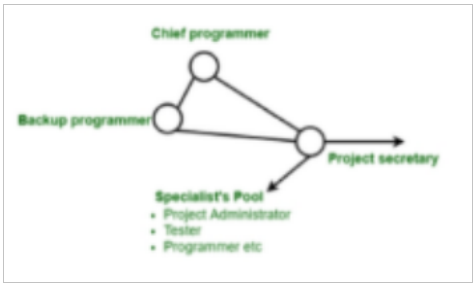
This team organization is composed of a small team consisting the following team members :

The Chief programmer: It is the person who is actively involved in the planning, specification and design process and ideally in the implementation process as well.

The project assistant: It is the closest technical co-worker of the chief programmer.

The project secretary: It relieves the chief programmer and all other programmers of administration tools.

Specialists: These people select the implementation language, implement individual system components and employ software tools and carry out tasks.



Matrix Team Organization:

In matrix team organization, people are divided into specialist groups. Each group has a manager.

Egoless Team Organization:

Egoless programming is a state of mind in which programmer are supposed to separate themselves from their product. In this team organization goals are set and decisions are made by group consensus. Here group, 'leadership' rotates based on tasks to be performed and differing abilities of members. In this organization work products are discussed openly and all freely examined all team members. There is a major risk which such organization, if teams are composed of inexperienced or incompetent members.

Democratic Team Organization:

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

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

Risk Management

Risk Management is an important part of project planning activities. It involves identifying and estimating the probability of risks with their order of impact on the project.

Risk Management Steps

There are some steps that need to be followed in order to reduce risk. These steps are as follows:

1. Risk Identification

Risk identification involves brainstorming activities. it also involves the preparation of a risk list. Brainstorming is a group discussion technique where all the stakeholders meet together. This technique produces new ideas and promotes creative thinking. Preparation of risk list involves identification of risks that are occurring continuously in previous software projects. The following risks are identified

- ✔ Project risks
- ✔ Technical risks
- ✔ Business risks

2. Risk Assessment (Analysis and Prioritization)

During the risk assessment process, an enterprise identifies potential risks that could harm its ability to operate. The next step in the risk management process after risk identification is risk analysis. This is where a company categorizes the potential risks and assigns a risk level to each one based on the likelihood that it will occur as well as its impact on the business.

- It is a process that consists of the following steps:
- Identifying the problems causing risk in projects
- Identifying the probability of occurrence of problem
- Identifying the impact of problem

Prepare a table consisting of all the values and order risk on the basis of risk exposure factor

Risk No	Problem	Probability of occurrence of problem	Impact of problem	Risk exposure	Priority
R1	Issue of incorrect	2	2	4	10

	Password				
R2	Testing reveals a lot of defects	1	9	9	7
R3	Design is not robust	2	7	14	5

3. Risk Avoidance and Mitigation

Risk mitigation is another step in the risk management process. It's not enough for an organization to assess and analyze the various types of risk, it also has to do something about those risks. There are a number of risk mitigation strategies a company can implement to deal with the various types of risk, including risk avoidance and risk reduction. The purpose of this technique is to altogether eliminate the occurrence of risks. so, the method to avoid risks is to reduce the scope of projects by removing non-essential requirements. There are three main strategies for risk containment:

- ✔ Avoid the risk (process-related, product-related and technology-related risks)
- ✔ Transfer the risk
- ✔ Risk reduction

4. Risk Monitoring

In this technique, the risk is monitored continuously by reevaluating the risks, the impact of risk, and the probability of occurrence of the risk. This ensures that:

- ✔ Risk has been reduced
- ✔ New risks are discovered
- ✔ Impact and magnitude of risk are measured

COCOMO Model

COCOMO (Constructive Cost Model) is a regression model based on LOC, i.e, number of Lines of Code. It is a procedural cost estimate model for software projects and often used as a process of reliably predicting the various parameters associated with making a project such as size, effort, cost, time and quality. It was proposed by Barry Boehm in 1981, ground work started in 1970 and is based on the study of 63 projects, which make it one of the best-documented models.

The key parameters which define the quality of any software products, which are also an outcome of the COCOMO are primarily Effort & Schedule:

Effort: Amount of labor that will be required to complete a task. It is measured in person

-months units.

Schedule: Simply means the amount of time required for the completion of the job, which is, of course, proportional to the effort put. It is measured in the units of time such as weeks, months. Different models of COCOMO have been proposed to predict the cost estimation at different levels, based on the amount of accuracy and correctness required. All of these models can be applied to a variety of projects, whose characteristics determine the value of constant to be used in subsequent calculations. These characteristics pertaining to different system types are mentioned below. Boehm's definition of organic, semidetached and embedded systems.

Organic: A software project is said to be an organic type if the team size required is adequately small, the problem is well understood and has been solved in the past and also the team members have a nominal experience regarding the problem.

Semi-detached: A software project is said to be a Semi-detached type if the vital characteristics such as team-size, experience, knowledge of the various programming environment lie in between that of organic and Embedded. The projects classified as Semi-Detached are comparatively less familiar and difficult to develop compared to the organic ones and require more experience and better guidance and creativity. Eg: Compilers or different Embedded Systems can be considered of Semi-Detached type.

Embedded: A software project with requiring the highest level of complexity, creativity, and experience requirement fall under this category. Such software requires a larger team size than the other two models and also the developers need to be sufficiently experienced and creative to develop such complex models. All the above system types utilize different values of the constants used in Effort Calculations.

Types of COCOMO Models

COCOMO consists of a hierarchy of three increasingly detailed and accurate forms. Any of the three forms can be adopted according to our requirements. These are types of COCOMO model:

- Basic COCOMO Model
- Intermediate COCOMO Model
- Detailed COCOMO Model

Basic COCOMO can be used for quick and slightly rough calculations of Software Costs. Its accuracy is somewhat restricted due to the absence of sufficient factor

considerations. Intermediate COCOMO takes these Cost Drivers into account and Detailed COCOMO additionally accounts for the influence of individual project phases, i.e in case of Detailed it accounts for both these cost drivers and also calculations are performed phase wise henceforth producing a more accurate result. These two models are further discussed below.

Estimation of Effort: Calculations

1. **Basic Model**

   $E = a(KLOC)^b$

   $time = c(Effort)^d$

   Person required = Effort/ time

The above formula is used for the cost estimation of for the basic COCOMO model, and also is used in the subsequent models. The constant values a, b, c and d for the Basic Model for the different categories of system:

Software Projects	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Semi Detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

The effort is measured in Person-Months and as evident from the formula is dependent on Kilo-Lines of code. The development time is measured in Months. These formulas are used as such in the Basic Model calculations, as not much consideration of different factors such as reliability, expertise is taken into account, henceforth the estimate is rough.

2. **Intermediate Model**

The basic COCOMO model assumes that the effort is only a function of the number of lines of code and some constants evaluated according to the different software system. However, in reality, no system's effort and schedule can be solely calculated on the basis of Lines of Code. For that, various other factors such as reliability, experience, Capability. These factors are known as Cost Drivers and the Intermediate Model utilizes 15 such drivers for cost estimation. Classification of Cost Drivers and their attributes:

(i) Product attributes:

- Required software reliability extent
- Size of the application database

- The complexity of the product

(ii) Hardware attributes:

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

(iii) Personnel attributes:

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

(iv) Project attributes:

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

The Intermediate COCOMO formula now takes the form:

$E= (a(KLOC)^b) * EAF$

The values of a and b in case of the intermediate model are as follows:

Software Projects	a	b
Organic	3.2	1.05
Semi Detached	3.0	1.12
Embeddedc	2.8	1.20

3. Detailed Model:

Detailed COCOMO incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step of the software engineering process. The detailed model uses different effort multipliers for each cost driver attribute. In detailed COCOMO, the whole software is divided into different modules and then we apply COCOMO in different modules to estimate effort and then sum the effort.

The Six phases of detailed COCOMO are:

- ✔ Planning and requirements
- ✔ System design

- ✔ Detailed design
- ✔ Module code and test
- ✔ Integration and test
- ✔ Cost Constructive model

The effort is calculated as a function of program size and a set of cost drivers are given according to each phase of the software lifecycle.

Software Configuration Management

In Software Engineering, 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. The primary reasons for implementing technical software configuration management system are:

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

Necessity of Software Configuration Management

There are several reasons for putting an object under configuration management. The following are some important problems that can crop up used SCM:

- ✔ Inconsistency problem when the objects are replicated
- ✔ Problems associated with concurrent access
- ✔ Providing a stable development environment
- ✔ System accounting and maintaining status information
- ✔ Handling variants

Configuration Management Activities

There are two principal activities

- Configuration identification
- Configuration control

Configuration identification is a method of determining the scope of the software system management. Project manager normally classify the objects associated with a software development into three main categories controlled, pre-controlled and uncontrolled. Controlled objects are those that are already under configuration control. Pre-controlled objects are not yet under configuration control, but will eventually be under configuration control. Uncontrolled objects are not subject to configuration control. Controllable objects include both controlled and pre-controlled objects. Typical controlled objects include:

- o Requirements Specification Document
- o Design documents
- o Tools used to build the system
- o Source code for each module
- o Test cases
- o Problem reports

Configuration Control: It is the process of managing changes to controlled objects. The configuration control part of a configuration management system that most directly affects the day-to-day operations.

Source Code Control System (SCCS) and Revision Control System (RCS)

SCCS and RCS are two popular configuration tools available on most UNIX system. SCCS and RCS can be used for controlling and managing different versions of text files. It does not handle binary files. It provides an efficient way of storing versions that minimizes the amount of occupied disk space.

The Nature of Software

Today, software can be considered in a dual role. It is a product, and at the same time, the vehicle for delivering a product.

Software as a product

- It delivers the computing potential embodied by computer hardware.
- Network of computers that are accessible by local hardware.
- It resides within a mobile phone or operates inside a mainframe computer.
- S/w is information transformer (producing, managing, acquiring, modifying, displaying or transmitting information that can be as simple as a single bit or as a complex).

Software as a vehicle

- Used to deliver the product.
- S/w acts as the basis for the control of the computer (operating systems).
- The communication of information (networks).
- The creation and control of other programs (s/w tools and environment).

Defining Software

Software is defined as

Instructions (Computer programs) that when executed provide desired features, function and performance.

Data structures that enable the programs to adequately manipulate information.

Documents descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

Characteristics of software

- Software is developed or engineered; it is not manufactured in the classical sense.
- Software does not “wear out”. However, it deteriorates due to change.
- Although the industry is moving towards component-based construction, most software continues to be custom built (Software is complex).
- Software is custom built rather than assembling existing components.

Legacy software

Legacy software is older programs that are developed decades ago. Examples of legacy software's are COBOL, ALP, ADA, BASIC, FORTRAN, PASCAL and LOTUS so on. The quality of legacy software is poor because it has inextensible design, convoluted code, poor and nonexistent documentation, test cases and results that are not achieved.

Non-Legacy software

Non-Legacy software is new programs that are developing present days. Examples of non-legacy software's are C, C++, JAVA, PYTHON so on. The quality of non-legacy software is high using these different types of applications are developed, it has extensible design, complex code, high existent documentation, test cases and results that are achieved.

Software Application Domains

Systems software: System software is a collection of programs written to service other programs. Examples compilers, editors, drivers so on.

Application software: Application software consists of standalone programs that solve a specific business need.

Engineering and scientific software: This is the software using "number crunching" algorithms for different scientific software applications.

Embedded software: it resides within a product or system and is used to implement and control features and functions for the end user and for the system itself.

Product-line software: it is designed to provide a specific facility for use by many different customers. Examples inventory control products, word processing, spreadsheets so on.

Web-based software: it is used to develop and retrieved web pages by a browser. A browser is software that includes executable instructions. Example HTML, Perl so on.

Artificial intelligence software: it uses non-numerical algorithms to solve complex problems. Example pattern recognition.

The Unique Nature of WebApps

In the early days the www, websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics. As time passed the augmentation of HTML by developing tools enabled engineers to provide computing capabilities along with informational content.

Today web-based system and applications were burn that is web apps have evaluated

into sophisticated computing tools that not only provide standalone function to the end user, but also have been integrated with corporate databases and business applications. That means web-based system and application involve a mixture between print publishing and software development technology. The following attributes are encountered in the vast majority of web apps.

Network intensiveness: A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable world wide access and communication or more limited access and communication.

Concurrency: A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.

Unpredictable load: The number of users of the WebApp may vary by orders of magnitude from day to day.

Performance: If a WebApp user must wait too long, may decide to go elsewhere.

Availability: Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a 24/7/365 basis.

Data driven: The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user.

Content sensitive: The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.

Continuous evolution: Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously.

Immediacy: Although immediacy – the compelling need to get software to market quickly - is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.

Security: Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure modes of data transmission, strong security measures must be implemented throughout the infrastructure that supports a WebApp and within the application itself.

Aesthetics: An undeniable part of the appeal of a WebApp is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

Software Myths

Software myths are erroneous beliefs about software and the process that is used to build it can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious. Today, most knowledgeable software engineering professionals recognize myths for what they are misleading attitudes that have caused serious problems for managers and practitioner's alike, old attitudes and habits are difficult to modify, and remnants of software myths remain. Software myths are divided into three types

1. Management myths: Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality.

Myths

- We already have a book that's full of standards and procedures for building software.
- If we get behind schedule, we can add more programmers and catch up.
- If I decide to outsource the software project to a third party, I can just relax and let that firm build it.

2. Customer myths: A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract.

Myths

- A general statement of objectives is sufficient to begin writing programs, fill in the details later.
- Software requirements continually change, but change can be easily accommodated because software is flexible.

3. Practitioner's myths: Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form.

Myths

- Once write the program and get it to work, our job is done.
- Until I get the program "running" I have no way of assessing its quality.
- The only deliverable work product for a successful project is the working

program.

- Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

Requirements Gathering and Analysis

The requirements gathering and analysis phase is the first phase of the SDLC. It is also called as requirements eliciting. It combines elements of problem solving, elaboration, negotiation and specification. Eliciting requirements are task of communicating with customers and users to determine what their requirements are. This is sometimes also called requirements gathering. Requirements gathering is a fundamental part of any software development project. These are things like "User wants to do X. How is this achieved?" In effect, Requirements Gathering is the process of generating a list of requirements (functional, system, technical, etc.) from all the stakeholders (customers, users, vendors, IT staff) that will be used as the basis for the formal definition of what the project is.

Analyzing requirements determining whether the stated requirements are unclear, incomplete, ambiguous, or contradictory and then resolving these issues. Requirement analysis is also known as Requirement Engineering. It is the process of defining user expectations for a new software being built or modified. In software engineering, it is sometimes referred to loosely by names such as requirements gathering or requirements capturing. Requirement's analysis encompasses those tasks that go into determining the needs or conditions to meet for a new or altered product or project, taking account of the possibly conflicting requirements of the various stakeholders, analyzing, documenting, validating and managing software or system requirements.

Software Requirement Specification (SRS)

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

modifications that is needed to be done to increase quality of product and to satisfy customer's demand.

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

- Introduction
  - Purpose of this document
  - Scope of this document
  - Overview
- General Description
- Functional Requirements
- Interface Requirements
- Performance Requirements
- Design Constraints
- Non-Functional Attributes
- Preliminary Schedule and Budget
- Appendices

Introduction:

Purpose: At first, main aim of why this document is necessary and what's purpose of document is explained and described.

Scope: In this, overall working and main objective of document and what value it will provide to customer is described and explained. It also includes a description of development cost and time required.

Overview: In this, description of product is explained. It's simply summary or overall review of product.

General Description:

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

Functional Requirements:

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

Interface Requirements:

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

Performance Requirements:

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

Design Constraints:

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

Non-Functional Attributes:

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

Preliminary Schedule and Budget:

In this, initial version and budget of project plan are explained which include overall time duration required and overall cost required for development of project.

Appendices:

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

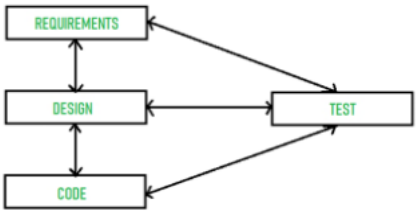
Traceability

Traceability comprises of two words i.e. trace and ability. Trace means to find someone or something and ability means to a skill or capability or talent to do something. Therefore, traceability simply means the ability to trace the requirement, to provide better quality, to find any risk, to keep and verify the record of history and production of an item or product by the means of documented identification. Due to this, it's easy for the suppliers to reduce any risk or any issue if found and to improve the quality of the item or product. So, it's important to have traceability rather than no traceability. Using traceability, finding requirements, and any risk to improve the quality of the product becomes very easy. There are various types of traceability given below:

Source traceability: These are basically the links from requirement to stakeholders who propose these requirements.

Requirement's traceability: These are the links between dependent requirements.

Design traceability: These are the links from requirement to design.



Traceability matrix is generally used to represent the information of traceability. For mentioning the traceability of small systems usually the traceability matrix is maintained. If one requirement is dependent upon another requirement then in that row-column cell 'D' is mentioned and if there is a weak relationship between the requirements than corresponding entry can be denoted by 'R'. For example:

Requirement ID	A	B	C	D	E	F
A		D			R	
B			D			
C				R		
D			D			R
E						
F	R			D		

Characteristics of a good SRS

The following are the characteristics of a good SRS document:

Correctness: User review is used to ensure the correctness of requirements stated in the SRS. SRS is said to be correct if it covers all the requirements that are actually expected from the system.

Completeness: Completeness of SRS indicates every sense of completion including the numbering of all the pages, resolving the to be determined parts to as much extent as possible as well as covering all the functional and non-functional requirements properly.

Consistency: Requirements in SRS are said to be consistent if there are no conflicts

between any set of requirements. Examples of conflict include differences in terminologies used at separate places, logical conflicts like time period of report generation, etc.

Unambiguousness: A SRS is said to be unambiguous if all the requirements stated have only 1 interpretation. Some of the ways to prevent unambiguousness include the use of modelling techniques like ER diagrams, proper reviews and buddy checks, etc.

Ranking for importance and stability: There should a criterion to classify the requirements as less or more important or more specifically as desirable or essential. An identifier mark can be used with every requirement to indicate its rank or stability.

Modifiability: SRS should be made as modifiable as possible and should be capable of easily accepting changes to the system to some extent. Modifications should be properly indexed and cross-referenced.

Verifiability: A SRS is verifiable if there exists a specific technique to quantifiably measure the extent to which every requirement is met by the system. For example, a requirement starting that the system must be user-friendly is not verifiable and listing such requirements should be avoided.

Traceability: One should be able to trace a requirement to design component and then to code segment in the program. Similarly, one should be able to trace a requirement to the corresponding test cases.

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

Testability: A SRS should be written in such a way that it is easy to generate test cases and test plans from the document.

Understandable by the customer: An end user maybe an expert in his/her specific domain but might not be an expert in computer science. Hence, the use of formal notations and symbols should be avoided to as much extent as possible. The language should be kept easy and clear.

Right level of abstraction: If the SRS is written for the requirements phase, the details should be explained explicitly. Whereas, for a feasibility study, fewer details can be used. Hence, the level of abstraction varies according to the purpose of the SRS.

Representing Complex Requirements using Decision Tables and Decision

Trees

A software system often is governed by complex logic, with various combinations of conditions and actions leading to different system behaviors. For example, if the driver presses the accelerate button on a car's cruise control system and the car is currently cruising, the system increases the car's speed. If the car isn't cruising, though, the system shouldn't do anything. Developers must base their work on a set of functional requirements that describe what the system should do under all possible combinations of both normal and error conditions. However, it's easy to overlook a specific condition combination, which then results in a missing requirement. These knowledge gaps are hard to spot by reviewing a specification written in natural language, with all its bulkiness and ambiguities.

Decision tables and decision trees are two effective techniques for representing how a system should behave when complex logic and decisions come into play. These techniques are valuable when defining functional requirements and business rules. They're also a great aid for efficient test planning.

1. Decision Tables:

Decision table is just a tabular representation of all conditions and actions. Decision trees are always used whenever the process logic is very complicated and involves multiple conditions. The main components used for the formation of Data Table are Conditions Stubs, Action Stubs, and rules.

2. Decision Trees:

Decision tree is a graph which always uses a branching method in order to demonstrate all the possible outcomes of any decision. Decision Trees are graphical and shows better representation of decision outcomes. It consists of three nodes namely Decision Nodes, Chance Nodes and Terminal Nodes.

Difference between Decision Tables and Decision Trees

Decision Tables	Decision Trees
Decision Tables are tabular representation of conditions and actions.	Decision Trees are graphical representation of every possible outcome of a decision.
We can derive decision table from decision tree.	We cannot derive decision tree from decision table.

It helps to clarify the criteria.	It helps to take into account the possible relevant outcomes of decision.
In Decision Tables, we can include more than one 'or' condition.	In Decision Trees, we cannot include more than one 'or' condition.
It is used when there are small number of properties.	It is used when there are more number of properties.
It is used for simple logic only.	It can be used for complex logic as well.
It is constructed of rows and tables.	It is constructed of branches and nodes.

Overview of Formal System Development Techniques

Formal methods are system design techniques that use rigorously specified mathematical models to build software and hardware systems. In contrast to other design systems, formal methods use mathematical proof as a complement to system testing in order to ensure correct behavior. As systems become more complicated, and safety becomes a more important issue, the formal approach to system design offers another level of insurance.

Formal methods are techniques used by software engineers to design safety-critical systems and their components. In software engineering, they are techniques that involve mathematical expressions to model "abstract representation" of the system.

Such models are subject to proof-check (Formal Specification) with regards to stability, cohesion and reliability. Proving validation is a core process for evaluating models using automatic theorem proofs. This is based on a set of mathematical formulas to be proven called proof obligations (Formal Verification). This allows identification of potential flaws earlier in the design stage, to prevent from "bricking" expensive systems later when placed into exploitation. Standard development techniques revolve around the following phases:

- ✔ Requirement's engineering
- ✔ Architecture design
- ✔ Implementation
- ✔ Testing
- ✔ Maintenance

- ✔ Evolution

There are notable differences between standard and formal software development methods. Formal methods are somewhat supporting tools. Here, the reliability of mathematics improves software production quality at any stage. They are not necessarily there to implement data processing. Choice of programming language is irrelevant. Instead, it creates a 'bridge' between modelled concepts and the environment towards final software implementation.

- Syntactic domains
- Semantic domains
- Satisfaction relation

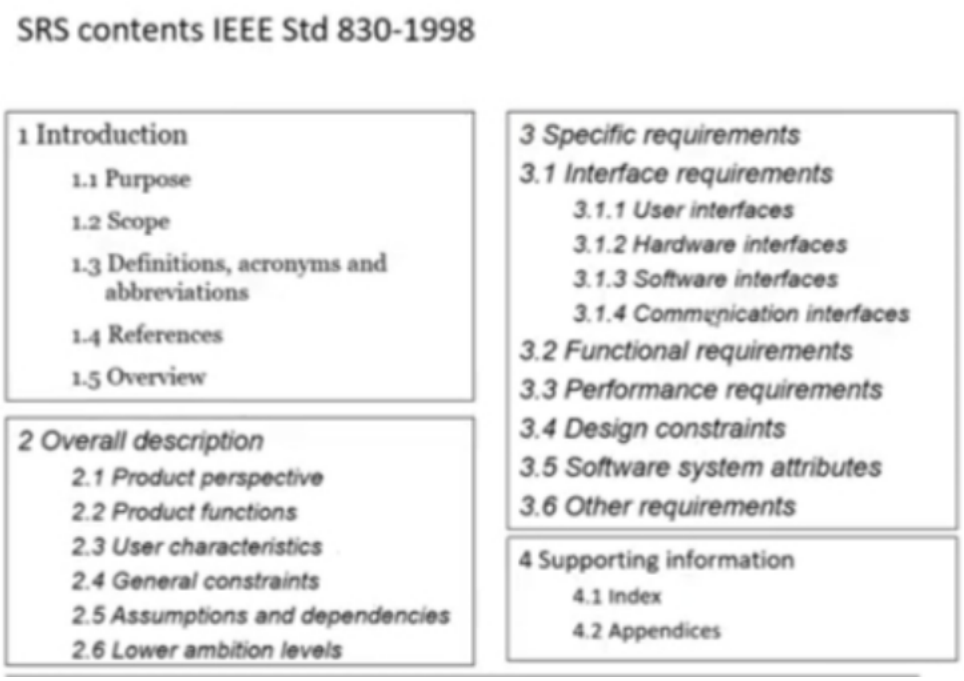
Axiomatic Specification

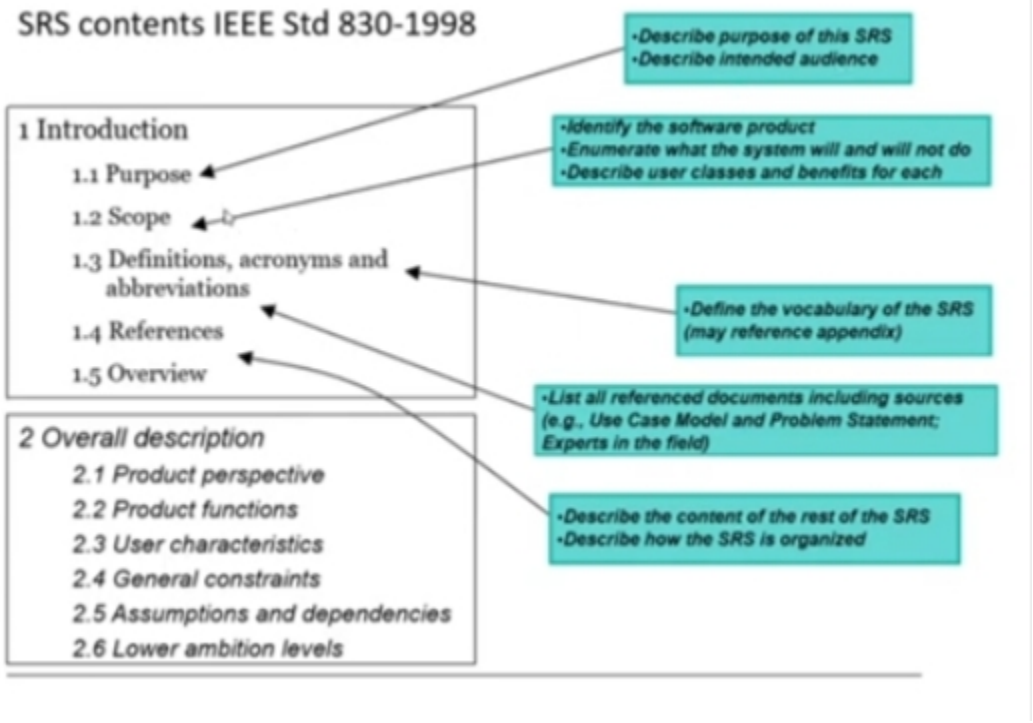
The Axiomatic Specification is a formal specification defining the semantics of functions of objects by a description of the relations between different objects and functions. The description is made by axioms (predicate-logical formula). In the Axiomatic Specification of a system, first-order logic is used to write the pre-and post-conditions to specify the operations of the system in the form of axioms. The pre-conditions capture the conditions that must be satisfied before an operation can successfully be invoked. In essence, the pre-conditions capture the requirements on the input parameters of a function. The post-conditions are the conditions that must be satisfied when function post-conditions are essentially constraints on the results produced for the function execution to be considered successful.

Algebraic Specification

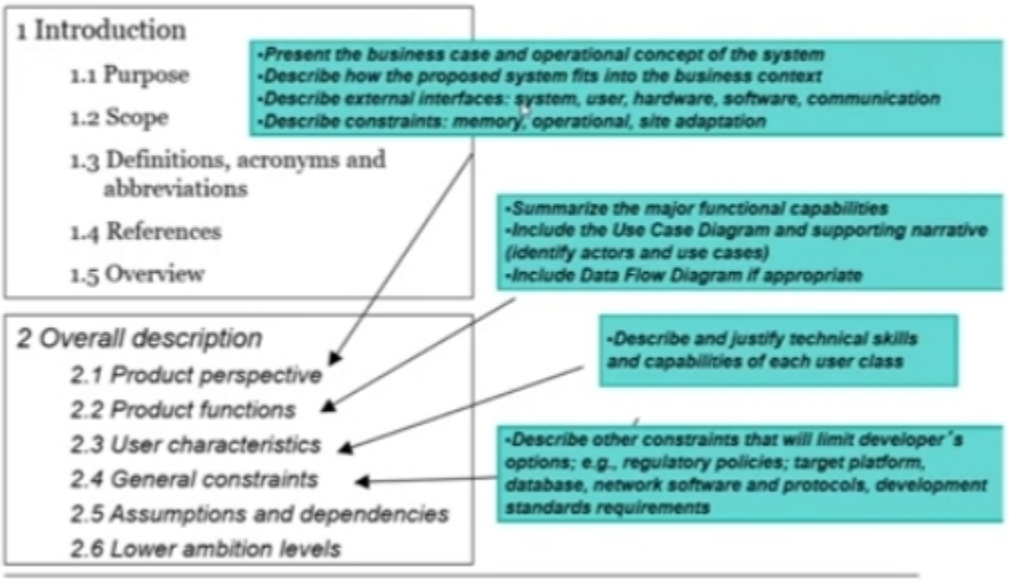
In the Algebraic Specification technique, an object class or type is specified in terms of relationships existing between the operations defined on that type. It was first brought into prominence by Guttag (1980-1985) in the specification of abstract data types. Various notations of algebraic specifications have evolved, including those based on OBJ and Larch languages. The Algebraic Specification describes functions in the form of an algebra. An algebra consists of a signature and the axioms. The signature consists of a family of object sets (carrier sets) and a number of operations in this carrier set with their functionality. The axioms describe the characteristics of this operations by means of algebraic equations.

IEEE 830 Guidelines





SRS contents IEEE Std 830-1998



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:

- Correctness
- Understandability
- Efficiency
- Maintainability

Correctness: 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.

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.

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.

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

The purpose of Design phase in the Software Development Life Cycle is to produce a solution to a problem given in the SRS (Software Requirement Specification) document. The output of the design phase is Software Design Document (SDD).

Coupling: Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.

Types of Coupling

1. **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent to each other and communicating through data. Module communications don't contain tramp data. Example-customer billing system.
2. **Stamp Coupling:** In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors- this choice made by the insightful designer, not a lazy programmer.
3. **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.
4. **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.
5. **Common Coupling:** The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So, it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses and reduced

maintainability.

6. **Content Coupling:** In a content coupling, one module can modify the data of another module or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

Cohesion: Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.

Types of Cohesion

1. **Functional Cohesion:** Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.
2. **Sequential Cohesion:** An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.
3. **Communicational Cohesion:** Two elements operate on the same input data or contribute towards the same output data. Example- update record in the database and send it to the printer.
4. **Procedural Cohesion:** Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.
5. **Temporal Cohesion:** The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time-span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at unit time.
6. **Logical Cohesion:** The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.
7. **Coincidental Cohesion:** The elements are not related(unrelated). The elements

have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion. Ex- print next line and reverse the characters of a string in a single component.

Control Hierarchy

Control hierarchy, also called program structure, represents the organization of program components (modules) and implies a hierarchy of control. It does not represent procedural aspects of software such as sequence of processes, occurrence or order of decisions, or repetition of operations; nor is it necessarily applicable to all architectural styles.

Different notations are used to represent control hierarchy for those architectural styles that are amenable to this representation. The most common is the treelike diagram that represents hierarchical control for call and return architectures. However, other notations, such as Warnier-Orr and Jackson diagrams may also be used with equal effectiveness. In order to facilitate later discussions of structure, we define a few simple measures and terms. Referring to figure, depth and width provide an indication of the number of levels of control and overall span of control, respectively. Fan-out is a measure of the number of modules that are directly controlled by another module. Fan-in indicates how many modules directly control a given module.

The control relationship among modules is expressed in the following way: A module that controls another module is said to be superordinate to it, and conversely, a module controlled by another is said to be subordinate to the controller . For example, referring to figure, module M is superordinate to modules a, b, and c. Module h is subordinate to module e and is ultimately subordinate to module M. Width-oriented relationships (e.g., between modules d and e) although possible to express in practice, need not be defined with explicit terminology.

The control hierarchy also represents two subtly different characteristics of the software architecture: visibility and connectivity. Visibility indicates the set of program components that may be invoked or used as data by a given component, even when this is accomplished indirectly. For example, a module in an object-oriented system may have access to a wide array of data objects that it has inherited, but makes use of only a small number of these data objects. All of the objects are visible to the module. Connectivity indicates the set of components that are directly invoked or used as data

by a given component. For example, a module that directly causes another module to begin execution is connected to it.

Layering (or) Layered Design

In layered design solution, the modules are arranged into several layers based on their call relationships. A module is allowed to call only the modules that are at a lower layer. A layered technology, to develop software need to go from one layer to another. All layers are connected and each layer demands the fulfillment the previous layer.

Concepts and Terminologies of Layered Design

The following are some important concepts and terminologies with a layered design

Superordinate and Subordinate Modules: In a control hierarchy, a module that controls another module is said to be superordinate to it. A module controlled by another module is said to be subordinate to the controller.

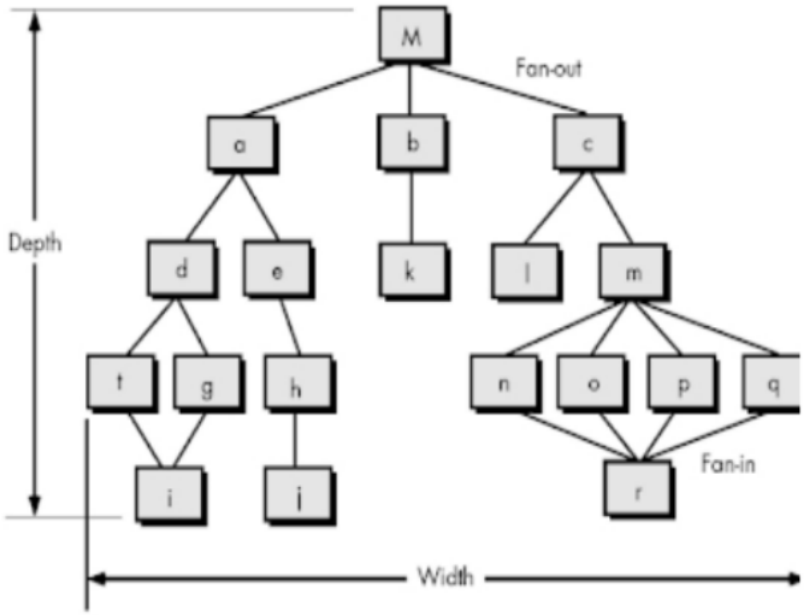
Visibility: A module B is said to be visible to another module A, if A directly calls B. thus, only the immediately lower layer modules are said to be visible to a module.

Control Abstraction: In a layered design, a module should only invoke the functions of the modules that are in the immediately below it. In other words, the modules at the higher layers, should not visible to modules at lower layers. This is called as control abstraction.

Depth and Width: Depth and Width of a control hierarchy prove an indication of the number of layers and overall span of control respectively.

Fan-out: Fan-out is a measure of the number of modules that are directly controlled by a given module.

Fan-in: Fan-in indicates the number of modules that directly invoke a given module. High fan-in represents code reuse and is I general, description in a good design.



Software Design Approaches

There are two fundamentally different approaches to software design. Function-oriented design and Object-oriented design. Through these two design approaches are completely different, they are complementary rather then competing techniques. Object-oriented approach is relatively newer technology and is still evolving. Function-oriented is developed technology and has long following.

Function Oriented Design

In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions. Function oriented design inherits some properties of structured design where divide and conquer methodology is used. This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation. These functional modules can share information among themselves by means of information passing and using information available globally.

Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state. The following are the salient features of the function-oriented design approach

- Top-Down Decomposition
- Centralized System State

Object-Oriented Design

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategies focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities. The important concepts of Object-Oriented Design:

Objects - All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.

Classes: A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object. In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.

Encapsulation: In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.

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

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

Object-Oriented Design Vs Function-Oriented Design

Object-oriented Design	Functional-oriented design
The basic abstractions are not the real world functions but are the data abstraction where the real world entities are represented.	The basic abstractions, which are given to the user, are real world functions.
Functions are grouped together on the basis of the data they operate since the classes are associated with their methods.	Functions are grouped together by which a higher level function is obtained.an eg of this technique isSA/SD
In this approach the state information is not represented in a centralized memory but is implemented or distributed among the objects of the system.	In this approach the state information is often represented in a centralized shared memory.
OOD approach is mainly used for evolving systems which mimics a business process or business case.	FOD approach is mainly used for computation sensitive application.
we decompose in class level	we decompose in function/procedure level
Bottom up approach	Top down Approach
Object-oriented design is the discipline of defining the objects and their interactions to solve a problem that was identified and documented during object-oriented analysis.	It views the system as Black Box that performs high level function and later decomposes it into a detailed function so to be mapped to modules.
Begins by identifying objects and classes.	Begins by considering the use case diagram and Scenarios.

Overview of SA/SD Methodology

Structured Analysis and Structured Design (SA/SD) is a diagrammatic notation that is designed to help people understand the system. The basic goal of SA/SD is to improve quality and reduce the risk of system failure. It establishes concrete management specifications and documentation. It focuses on the solidity, pliability, and maintainability of the system. Basically, the approach of SA/SD is based on the Data Flow Diagram. It is easy to understand SA/SD but it focuses on well-defined system boundary whereas the JSD approach is too complex and does not have any graphical representation. SA/SD is combined known as SAD and it mainly focuses on the following 3 points:

- System

- Process
- Technology

SA/SD phases:

Analysis Phase: Analysis Phase involves data flow diagram, data dictionary, state transition diagram, and entity-relationship diagram.

Data Flow Diagram: In the data flow diagram, the model describes how the data flows through the system. We can incorporate the Boolean operators and & or link data flow when more than one data flow may be input or output from a process.

For example, if we have to choose between two paths of a process, we can add an operator or and if two data flows are necessary for a process, we can add an operator. The input of the process "check-order" needs the credit information and order information whereas the output of the process would be a cash-order or a good-credit-order.

Data Dictionary: The content that is not described in the DFD is described in the data dictionary. It defines the data store and relevant meaning. A physical data dictionary for data elements that flow between processes, between entities, and between processes and entities may be included. This would also include descriptions of data elements that flow external to the data stores.

A logical data dictionary may also be included for each such data element. All system names, whether they are names of entities, types, relations, attributes, or services, should be entered in the dictionary.

State Transition Diagram: State transition diagram is similar to the dynamic model. It specifies how much time the function will take to execute and data access triggered by events. It also describes all of the states that an object can have, the events under which an object changes state, the conditions that must be fulfilled before the transition will occur and the activities were undertaken during the life of an object.

ER Diagram: ER diagram specifies the relationship between data store. It is basically used in database design. It basically describes the relationship between different entities.

Design Phase: Design Phase involves structure chart and pseudocode.

Structure Chart: It is created by the data flow diagram. Structure Chart specifies how DFS's processes are grouped into tasks and allocate to the CPU. The structured chart

does not show the working and internal structure of the processes or modules and does not show the relationship between data or data-flows. Similar to other SASD tools, it is time and cost-independent and there is no error-checking technique associated with this tool. The modules of a structured chart are arranged arbitrarily and any process from a DFD can be chosen as the central transform depending on the analysts' own perception. The structured chart is difficult to amend, verify, maintain, and check for completeness and consistency.

Pseudocode: It is the actual implementation of the system. It is an informal way of programming that doesn't require any specific programming language or technology.

Structured Analysis

Analysts use various tools to understand and describe the information system. One of the ways is using structured analysis. Structured Analysis is a development method that allows the analyst to understand the system and its activities in a logical way. It is a systematic approach, which uses graphical tools that analyze and refine the objectives of an existing system and develop a new system specification which can be easily understandable by user. It has following attributes:

- It is graphic which specifies the presentation of application.
- It divides the processes so that it gives a clear picture of system flow.
- It is logical rather than physical i.e., the elements of system do not depend on vendor or hardware.
- It is an approach that works from high-level overviews to lower-level details.

Structured Analysis Tools

During Structured Analysis, various tools and techniques are used for system development. They are

- Data Flow Diagrams
- Data Dictionary
- Decision Trees
- Decision Tables
- Structured English
- Pseudocode

Data Flow Diagrams (DFD) or Bubble Chart: DFD is easy to understand and quite

effective when the required design is not clear and the user wants a notational language for communication. However, it requires a large number of iterations for obtaining the most accurate and complete solution. It is a technique developed by Larry Constantine to express the requirements of system in a graphical form. It shows the flow of data between various functions of system and specifies how the current system is implemented. It is an initial stage of design phase that functionally divides the requirement specifications down to the lowest level of detail. Its graphical nature makes it a good communication tool between user and analyst or analyst and system designer. It gives an overview of what data a system processes, what transformations are performed, what data are stored, what results are produced and where they flow.

Data Dictionary: A data dictionary is a structured repository of data elements in the system. It stores the descriptions of all DFD data elements that is, details and definitions of data flows, data stores, data stored in data stores, and the processes. A data dictionary improves the communication between the analyst and the user. It plays an important role in building a database. Most DBMSs have a data dictionary as a standard feature.

Decision Trees: Decision trees are a method for defining complex relationships by describing decisions and avoiding the problems in communication. A decision tree is a diagram that shows alternative actions and conditions within horizontal tree framework. Thus, it depicts which conditions to consider first, second, and so on. Decision trees depict the relationship of each condition and their permissible actions. A square node indicates an action and a circle indicates a condition. It forces analysts to consider the sequence of decisions and identifies the actual decision that must be made.

Decision Tables: Decision tables are a method of describing the complex logical relationship in a precise manner which is easily understandable. It is useful in situations where the resulting actions depend on the occurrence of one or several combinations of independent conditions. It is a matrix containing row or columns for defining a problem and the actions.

Components of a Decision Table

Condition Stub: It is in the upper left quadrant which lists all the condition to be checked.

Action Stub: It is in the lower left quadrant which outlines all the action to be carried out to meet such condition.

Condition Entry: It is in upper right quadrant which provides answers to questions asked in condition stub quadrant.

Action Entry: It is in lower right quadrant which indicates the appropriate action resulting from the answers to the conditions in the condition entry quadrant.

The entries in decision table are given by Decision Rules which define the relationships between combinations of conditions and courses of action.

Structured English: Structure English is derived from structured programming language which gives more understandable and precise description of process. It is based on procedural logic that uses construction and imperative sentences designed to perform operation for action. It is best used when sequences and loops in a program must be considered and the problem needs sequences of actions with decisions. It does not have strict syntax rule. It expresses all logic in terms of sequential decision structures and iterations.

Pseudocode: A pseudocode does not conform to any programming language and expresses logic in plain English. It may specify the physical programming logic without actual coding during and after the physical design. It is used in conjunction with structured programming. It replaces the flowcharts of a program.

Data Flow Diagram (DFD)

DFD is the abbreviation for Data Flow Diagram. The flow of data of a system or a process is represented by DFD. It also gives insight into the inputs and outputs of each entity and the process itself. DFD does not have control flow and no loops or decision rules are present. Specific operations depending on the type of data can be explained by a flowchart. Data Flow Diagram can be represented in several ways. The DFD belongs to structured-analysis modeling tools. Data Flow diagrams are very popular because they help us to visualize the major steps and data involved in software-system processes.

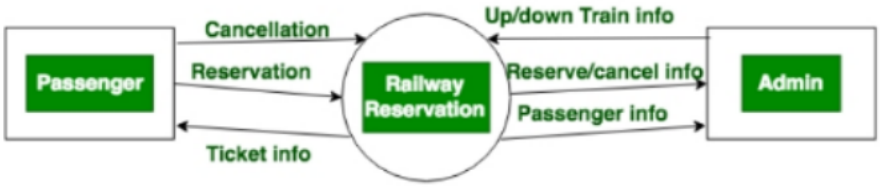
Components of DFD

Rectangle	: defines a source or destination of data
Circle or Square	: represents a process that transfers incoming data flow into outgoing data
Arrow	: identifies data flow, it is pipeline through which information flows.

Levels in Data Flow Diagrams (DFD)

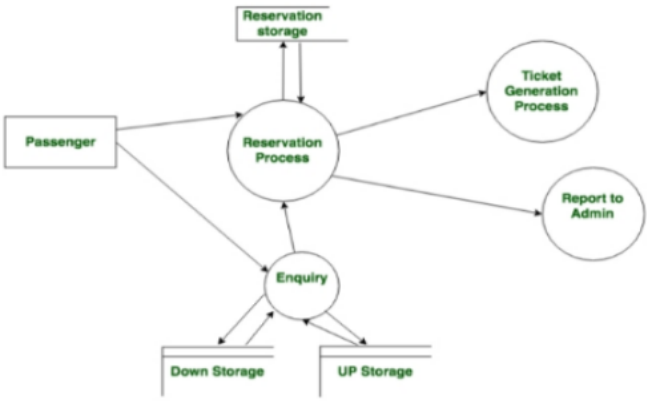
In Software engineering DFD (data flow diagram) can be drawn to represent the system of different levels of abstraction. Higher-level DFDs are partitioned into low levels-hacking more information and functional elements. Levels in DFD are numbered 0, 1, 2 or beyond. Here, we will see mainly 3 levels in the data flow diagram, which are: 0-level DFD, 1-level DFD, and 2-level DFD.

0-level DFD: It is also known as a context diagram. It's designed to be an abstraction view, showing the system as a single process with its relationship to external entities. It represents the entire system as a single bubble with input and output data indicated by incoming/outgoing arrows.



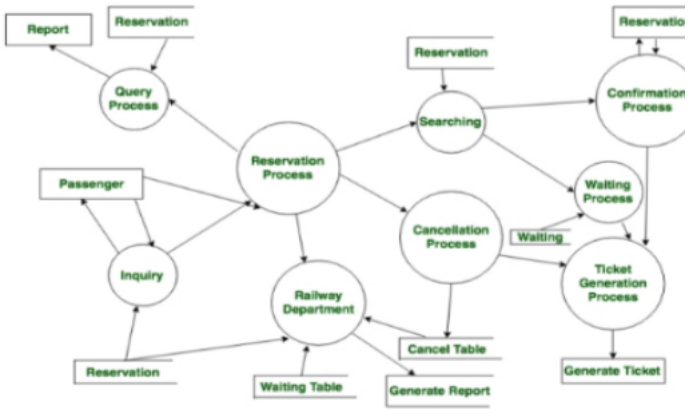
0-LEVEL DFD

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



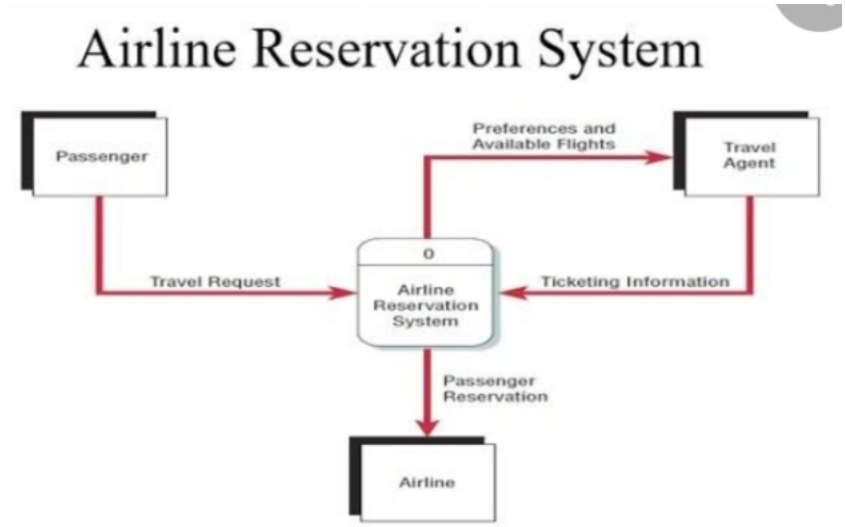
1-LEVEL DFD

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



2-LEVEL DFD

Extending DFD Techniques to Real Life System



Basic Object-Oriented Concepts

Object-oriented programming is a method used for designing a program using classes

and objects. Object-oriented programming is also called the core of java. Object-oriented programming organizes a program around objects and well-defined interfaces. This can also be characterized as data controlling for accessing the code. In this type of approach, programmers define the data type of a data structure and the operations that are applied to the data structure. This implies software development and maintenance by using some of the concepts:

- Object
- Class
- Data Abstraction
- Inheritance
- Polymorphism
- Encapsulation
- Message Passing
- Data hiding

Object

An object is a real-world element in an object–oriented environment that may have a physical or a conceptual existence. Objects are always called as instances of a class. Objects are created from class in java or any other languages. Objects are those that have state and behaviour. Objects are abstract data types (i.e., objects behaviour is defined by a set of values and operations).

Class

A class represents a collection of objects having same characteristic properties that exhibit common behavior. It gives the blueprint or description of the objects that can be created from it. Creation of an object as a member of a class is called instantiation. Thus, object is an instance of a class.

Data Abstraction

Data Abstraction for hiding the internal details or implementations of a function and showing its functionalities only. This is similar to the way you know how to drive a car without knowing the background mechanism. Or you know how to turn on or off a light using a switch but you don’t know what is happening behind the socket.

Inheritance

Inheritance is the mechanism that permits new classes to be created out of existing

classes by extending and refining its capabilities. The existing classes are called the base classes/parent classes/super-classes, and the new classes are called the derived classes/child classes/subclasses. The subclass can inherit or derive the attributes and methods of the super-class(es) provided that the super-class allows so. Besides, the subclass may add its own attributes and methods and may modify any of the super-class methods. Inheritance defines an “is – a” relationship.

Polymorphism

Polymorphism is originally a Greek word that means the ability to take multiple forms. In object-oriented paradigm, polymorphism implies using operations in different ways, depending upon the instance they are operating upon. Polymorphism allows objects with different internal structures to have a common external interface. Polymorphism is particularly effective while implementing inheritance.

Encapsulation

Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. It permits the elements of the class to be accessed from outside only through the interface provided by the class.

Message Passing

Any application requires a number of objects interacting in a harmonious manner. Objects in a system may communicate with each other using message passing. Suppose a system has two objects: obj1 and obj2. The object obj1 sends a message to object obj2, if obj1 wants obj2 to execute one of its methods.

Data hiding

Data hiding ensures exclusive data access to class members and protects objects integrity by preventing unintended or intended changes. Is also reduces system complexity for increased robustness by limiting interdependencies between software components.

UML Diagrams

Unified Modeling Language (UML) is a general-purpose modelling language. The main aim of UML is to define a standard way to visualize the way a system has been designed. It is quite similar to blueprints used in other fields of engineering. UML is not

a programming language; it is rather a visual language. We use UML diagrams to portray the behavior and structure of a system. UML helps software engineers, businessmen and system architects with modelling, design and analysis. The Object Management Group (OMG) adopted Unified Modelling Language as a standard in 1997. Its been managed by OMG ever since. International Organization for Standardization (ISO) published UML as an approved standard in 2005. UML has been revised over the years and is reviewed periodically.

Do we really need UML?

Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them. Businessmen do not understand code, so UML becomes essential to communicate with non-programmers, essential requirements, functionalities and processes of the system. A lot of time is saved down the line when teams are able to visualize processes, user interactions and static structure of the system. UML is linked with object-oriented design and analysis. UML makes the use of elements and forms associations between them to form diagrams.

Types of UML Diagrams

Diagrams in UML can be broadly classified as:

- Structural Diagrams
- Behavior Diagrams

Structural UML Diagrams

Capture static aspects or structure of a system. Structural Diagrams include: Component Diagrams, Object Diagrams, Class Diagrams and Deployment Diagrams.

Class Diagram: The most widely use UML diagram is the class diagram. It is the building block of all object-oriented software systems. We use class diagrams to depict the static structure of a system by showing system's classes, their methods and attributes. Class diagrams also help us identify relationship between different classes or objects.

Object Diagram: An Object Diagram can be referred to as a screenshot of the instances in a system and the relationship that exists between them. Since object diagrams depict behaviour when objects have been instantiated, we are able to study the behaviour of the system at a particular instant. An object diagram is similar to a class diagram

except it shows the instances of classes in the system. We depict actual classifiers and their relationships making the use of class diagrams. On the other hand, an Object Diagram represents specific instances of classes and relationships between them at a point of time.

Component Diagram: Component diagrams are used to represent the how the physical components in a system have been organized. We use them for modelling implementation details. Component Diagrams depict the structural relationship between software system elements and help us in understanding if functional requirements have been covered by planned development. Component Diagrams become essential to use when we design and build complex systems. Interfaces are used by components of the system to communicate with each other.

Deployment Diagram: Deployment Diagrams are used to represent system hardware and its software. It tells us what hardware components exist and what software components run on them. We illustrate system architecture as distribution of software artifacts over distributed targets. An artifact is the information that is generated by system software. They are primarily used when a software is being used, distributed or deployed over multiple machines with different configurations.

Behavior UML Diagrams

Capture dynamic aspects or behavior of the system. Behavior diagrams include: Use Case Diagrams, State Diagrams, Activity Diagrams and Interaction Diagrams.

State Machine Diagrams: A state diagram is used to represent the condition of the system or part of the system at finite instances of time. It's a behavioral diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as State machines and State-chart Diagrams. These terms are often used interchangeably. So, simply a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli.

Activity Diagrams: We use Activity Diagrams to illustrate the flow of control in a system. We can also use an activity diagram to refer to the steps involved in the execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram. An activity diagram focuses on condition of flow and the sequence in which it happens. We describe or depict what causes a particular event using an activity diagram.

Use Case Diagrams: Use Case Diagrams are used to depict the functionality of a system or a part of a system. They are widely used to illustrate the functional requirements of the system and its interaction with external agents(actors). A use case is basically a diagram representing different scenarios where the system can be used. A use case diagram gives us a high-level view of what the system or a part of the system does without going into implementation details.

Sequence Diagram: A sequence diagram simply depicts interaction between objects in a sequential order i.e., the order in which these interactions take place. We can also use the terms event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.

Interaction Diagrams: An Interaction Diagram models a sequence of actions and helps us simplify complex interactions into simpler occurrences. It is a mixture of activity and sequence diagrams and collaboration diagrams.

Communication Diagram: A Communication Diagram known as Collaboration Diagram in UML is used to show sequenced messages exchanged between objects. A communication diagram focuses primarily on objects and their relationships. We can represent similar information using Sequence diagrams; however, communication diagrams represent objects and links in a free form.

Timing Diagram: Timing Diagram are a special form of Sequence diagrams which are used to depict the behavior of objects over a time frame. We use them to show time and duration constraints which govern changes in states and behavior of objects.

Structured Design

Structured Design is a systematic methodology to determine design specification of software. Structured design is a conceptualization of problem into several well-organized elements of solution. It is basically concerned with the solution design. Benefit of structured design is, it gives better understanding of how the problem is being solved. Structured design also makes it simpler for designer to concentrate on the problem more accurately.

Structured design is mostly based on ‘divide and conquer’ strategy where a problem is

broken into several small problems and each small problem is individually solved until the whole problem is solved. The small pieces of problem are solved by means of solution modules. Structured design emphasis that these modules be well organized in order to achieve precise solution. These modules are arranged in hierarchy. They communicate with each other. A good structured design always follows some rules for communication among multiple modules, namely -

Cohesion: grouping of all functionally related elements.

Coupling: communication between different modules.

A good structured design has high cohesion and low coupling arrangements.

Detailed Design

Design is the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures. The detailed design may include:

- Decomposition of major system components into program units.
- Allocation of functional responsibilities to units.
- User interfaces
- Unit states and state changes
- Data and control interaction between units
- Data packaging and implementation, including issues of scope and visibility of program elements
- Algorithms and data structures.

Design Review

A design review is a milestone within a product development process whereby a design is evaluated against its requirements in order to verify the outcomes of previous activities and identify issues before committing to - and if need to be re-priorities further work. The ultimate design review, if successful, therefore triggers the product launch or product release. The conduct of design reviews is compulsory as part of design controls, when developing products in certain regulated contexts such as medical devices.

Software design reviews are a systematic, comprehensive, and well-documented inspection of design that aims to check whether the specified design requirements are

adequate and the design meets all the specified requirements. In addition, they also help in identifying the problems (if any) in the design process. IEEE defines software design review as 'a formal meeting at which a system's preliminary or detailed design is presented to the user, customer, or other interested parties for comment and approval.' These reviews are held at the end of the design phase to resolve issues (if any) related to software design decisions, that is, architectural design and detailed design (component-level and interface design) of the entire software or a part of it (such as a database).

Types of Software Design Reviews

Generally, the review process is carried out in three steps, which corresponds to the steps involved in the software design process.

Preliminary Design Review

During preliminary design review, the high-level architectural design is reviewed to determine whether the design meets all the stated requirements as well as the non-functional requirements. This review is conducted to serve the following purposes.

- To ensure that the software requirements are reflected in the software architecture
- To specify whether effective modularity is achieved
- To define interfaces for modules and external system elements
- To ensure that the data structure is consistent with the information domain
- To ensure that maintainability has been considered
- To assess the quality factors.

Critical Design Review

Once the preliminary design review is successfully completed and the customer(s) is satisfied with the proposed design, a critical design review is conducted. This review is conducted to serve the following purposes.

- To assure that there are no defects in the technical and conceptual designs
- To verify that the design being reviewed satisfies the design requirements established in the architectural design specifications
- To assess the functionality and maturity of the design critically
- To justify the design to the outsiders so that the technical design is more clear, effective and easy to understand.

Program Design Review

Once the critical design review is successfully completed, a program design review is conducted to obtain feedback before the implementation (coding) of the design. This review is conducted to serve the following purposes.

- To assure the feasibility of the detailed design
- To assure that the interface is consistent with the architectural design
- To specify whether the design is compatible to implementation language
- To ensure that structured programming constructs are used throughout
- To ensure that the implementation team is able to understand the proposed design.

Software Design Review Process

Design reviews are considered important as in these reviews the product is logically viewed as the collection of various entities/components and use-cases. These reviews are conducted at all software design levels and cover all parts of the software units.

Software Design Review Process

Design reviews are considered important as in these reviews the product is logically viewed as the collection of various entities/components and use-cases. These reviews are conducted at all software design levels and cover all parts of the software units. Generally, the review process comprises three criteria, as listed below.

Entry criteria: Software design is ready for review.

Activities: This criterion involves the following steps.

- ✔ Select the members for the software design review team, assign them their roles, and prepare schedules for the review.
- ✔ Distribute the software design review package to all the reviewing participants.
- ✔ Participants check the completeness and conformance of the design to the requirements in addition to the efficiency of the design.
- ✔ The software development manager obtains the approval of the software design from the software project manager.

Exit criteria: The software design is approved.

Evaluating Software Design Reviews

The software design review process is beneficial for everyone as the faults can be detected at an early stage, thereby reducing the cost of detecting errors and reducing

the likelihood of missing a critical issue. Every review team member examines the integrity of the design and not the persons involved in it (that is, designers), which in turn emphasizes that the common 'objective of developing a highly rated design is achieved.

Characteristics of Good User Interface

The following are the characteristics of good user interface

- Clear
- Concise
- Familiar
- Responsive
- Consistent
- Attractive
- Efficient
- Forgiving

Clear: Clarity is the most important element of user interface design. Indeed, the whole purpose of user interface design is to enable people to interact with your system by communicating meaning and function. If people can't figure out how your application works or where to go on your website they'll get confused and frustrated.

Concise: Clarity in a user interface is great, however, you should be careful not to fall into the trap of over-clarifying. It is easy to add definitions and explanations, but every time you do that you add mass. Your interface grows. Add too many explanations and your users will have to spend too much time reading through them.

Familiar: Familiar is just that: something which appears like something else you've encountered before. When you're familiar with something, you know how it behaves – you know what to expect. Identify things that are familiar to your users and integrate them into your user interface.

Responsive: Responsive means a couple of things. First of all, responsive means fast. The interface, if not the software behind it, should work fast. Waiting for things to load and using leggy and slow interfaces is frustrating. Seeing things load quickly, or at the very least, an interface that loads quickly (even if the content is yet to catch up) improves the user experience.

Consistent: Consistent interfaces allow users to develop usage patterns – they'll learn what the different buttons, tabs, icons and other interface elements look like and will recognize them and realize what they do in different contexts. They'll also learn how certain things work, and will be able to work out how to operate new features quicker, extrapolating from those previous experiences.

Attractive: This one may be a little controversial but I believe a good interface should be attractive. Attractive in a sense that it makes the use of that interface enjoyable. Yes, you can make your UI simple, easy to use, efficient and responsive, and it will do its job well – but if you can go that extra step further and make it attractive, then you will make the experience of using that interface truly satisfying.

Efficient: A user interface is the vehicle that takes you places. Those places are the different functions of the software application or website. A good interface should allow you to perform those functions faster and with less effort.

Forgiving: A forgiving interface is one that can save your users from costly mistakes. For example, if someone deletes an important piece of information, can they easily retrieve it or undo this action? When someone navigates to a broken or nonexistent page on your website, what do they see? Are they greeted with a cryptic error or do they get a helpful list of alternative destinations?

User Guidance and Online Help

BASIC CONCEPTS OF USER INTERFACE DESIGN

User Guidance and On-line Help

Users may seek help about the operation of the software any time while using the software. This is provided by the on-line help system.

On-line help system: A good online help system should keep track of what a user is doing while invoking the help system and provide the output message in a context-dependent way. It should take advantage of any graphics and animation characteristics of the screen.

Guidance messages: The guidance messages should be carefully designed to prompt the user about the next actions he might pursue, the current status of the system, the progress so far made in processing his last command, etc.

Error messages: Error messages are generated by a system either when the user commits some error or when some errors encountered by the system during processing due to some exceptional conditions, such as out of memory, communication link broken, etc.

Mode-based Vs Modeless

A mode is state or collection of states in which only a subset of all user interaction tasks can be performed. In a modeless interface, the same set of commands can be invoked at any time during the running of the software. Thus, a modeless interface has only a single mode and all the commands are available all the time during the operation of the software. On the other hand, in a mode-based interface different set for commands can be invoked depending on the mode in which the system.

User Interface Design

User interface is the front-end application view to which user interacts in order to use the software. It is also called as human-computer interface. It handles the interaction between the user and the system. It describes the way in which a human interacts with a machine. Though technically something as simple as a light switch could be considered an instrument of UI, most modern references relate to computers and other electronic devices. UI makes the exchange between users and machines possible. Without it, this vital form of communication ceases to exist. The software becomes more popular if its user interface is:

- Attractive

- Simple to use
- Responsive in short time
- Clear to understand
- Consistent on all interface screens

Types of User Interface

There are three types of user interface

- Command Line Interface
- Menu-driven Interface
- Graphical User Interface

Command Line Interface

The command line interface is no longer common as a form of basic user interface in everyday consumer products, but it is still in use under certain circumstances. Command Line Interface requires users to type appropriate instructions into the command line. The computer is commanded to first go to the required file or directory. From there, a whole host of commands become available, from retrieving files to running programs.

Command line elements

The different elements of the CLI are as follows -

Command Prompt: The command prompt displays the user context. Software system generates the command prompt.

Cursor: The position of the character that is being typed is represented by the small horizontal line or a vertical bar known as cursor. When something is written or deleted by the user, the cursor moves accordingly.

Command: The instruction that needs to be executed is known as command. On execution of the command, the output is displayed on the screen. Command prompt appears on the next line on production of the output.

Advantages

- Simple structure
- Minimal memory usage
- Great for slow-running computers, or those low on memory
- An expert CLI user can give commands and perform tasks much faster than when using an alternative UI type

Disadvantages

- Difficult to learn command language
- Complex for novice users
- Minimal error message information

Menu-Driven Interface

The menu-driven user interface provides you with a range of commands or options in the form of a list or menu displayed in full-screen, pop-up, pull-down, or drop-down. An ATM is an example of a menu-driven interface.

Advantages

- It is not necessary to remember a long list of manual commands
- Simple interface for novices
- Self-explanatory menu options

Disadvantages

- Slower for experienced users
- Limited menu options
- Often requires you to access multiple menu screens to perform simple functions

Graphical User Interface

The graphical user interface, or GUI, is the type of interface with which the majority of people are the most familiar. You interact with these interfaces by using a mouse, tack pad, or other peripheral to point and click on graphics or icons.

Advantages

- Self-explanatory
- Easy to use
- Memorizing command lists is not necessary
- Allows for running multiple applications, programs, and tasks simultaneously
- Solid support facilities
- The similar format among different programs adds familiarity
- WYSIWYG makes for easy design and formatting

Disadvantages

- Uses large amounts of memory – although this is less of a concern as computers get more powerful.

GUI Design Methodology

user is facilitated with the graphical means for communicating with the system and which is known as Graphical User Interface. Hardware and software combine together and constitute GUI. The software is interpreted by the user with the help of GUI. When compared with CLI, amount of the resources utilized is high for GUI. More complex GUI designs are developed by the designers and programmers for increasing the accuracy, efficiency and speed of the interface.

GUI Elements

The hardware or software system can be communicated through GUI by a means of different components. Different ways by which the user is enabled to interact and work with the system are made available by the components of the GUI. GUI system includes the following elements -

Window: The application contents that are displayed either as icons or as lists in a window. The structure of the file is demonstrated by a window. A window can be explored and a file ca ne easily found in the system. It is possible to minimize, resize or maximize the size of the window. They can be dragged and moved as desired on the screen. A window with another window of same application is known as child window.

Tabs: Users are enabled to open more than one document in the same window. This is facilitated by tabbed document interface. This feature is used by almost all the web browsers.

Menu: All the standard commands are collected, grouped together, arranged and are displayed at a place that is clearly visible inside the window. As desired, an option is provided to either hide or display the menu, by certain programming.

Icon: An application is replicated by small picture known as an icon. The application window can be visited and opened on clicking on this icon. All the programs installed on the system and the applications are displayed by using small pictures as Icons.

Cursor: Cursor include all the devices used for interaction such as mouse, digital pen, touch pad etc. The cursor acts according to the instruction of the hardware. The desired menu, window or an application can be selected by using a cursor.

Application Specific GUI Components

Some of the elements of components of GUI are specific to only GUI application. They

are as follows -

Application Window: The operating system provides the constructs for many of the uses of application windows. The custom created windows are used instead as they include the application content.

Dialogue Box: A message is communicated to the user such that the user needs to act upon, by using a small window known as a child window or a dialog box. For instance, in order to delete, the application will generate a dialog box and which proceeds with the process of deletion only upon the receipt of the confirmation by the user through the dialog box.

Text-Box: An area made available for the user to type the data.

Buttons: Some of the inputs are submitted to the software by using buttons.

Radio-button: For a specific input, when two options are made available, both of the options are displayed by Radio-button. This button enables to select only one.

Check-box: For all the options that are being selected by the user, a box is marked as checked.

List-box: All the available items are listed in the list-box. This enables the user to select from the list. The user is facilitated to select more than one item from the list. Other impressive GUI components are: Sliders, Combo-box, Data-grid, Drop-down list

What are the different activities performed by a User Interface Design?

User Interface design performs many activities. The design and implementation of GUI is similar to that of SDLC design. Among the different models such as Waterfall, Iterative or Spiral, any one of them can be chosen for implementing GUI. Some of the specific steps need to be satisfied or followed by any model that is selected for the design and the implementation of GUI. The steps are as follows -

GUI Requirement Gathering: All the functional and non-functional requirements of GUI need to be collected, gathered and maintained by the designers.

User Analysis: The user of the GUI is analyzed by the designer. As per the knowledge and level of competency of the user, the design is changed. Advanced GUI is implemented for the technical savvy user. More basic information on how to use the software is provided for the non-technical user.

Task Analysis: The task that is to be taken up the software is analyzed by the designer. In GUI the tasks are analyzed by breaking the major task into sub-tasks. The goals to be

achieved by the GUI are determined by these tasks. The content that has to pass through the GUI is determined by the information that flows in the sub-tasks.

GUI Design & Implementation: Considering the user environment, tasks and the knowledge about the requirements, GUI is designed and implemented. Initially GUI is implemented on a dummy project to testing.

Testing: GUI can be tested in different ways. Some of them include in-house verification, user involvement and introduction of beta version. The usability of GUI, the acceptance of GUI by the user and the GUI compatibility are Some of the aspects on which testing.

Coding Standards and Guidelines

Different modules specified in the design document are coded in the Coding phase according to the module specification. The main goal of the coding phase is to code from the design document prepared after the design phase through a high-level language and then to unit test this code. Good software development organizations want their programmers to maintain to some well-defined and standard style of coding called coding standards. They usually make their own coding standards and guidelines depending on what suits their organization best and based on the types of software they develop. It is very important for the programmers to maintain the coding standards otherwise the code will be rejected during code review.

Purpose of Coding Standards

- A coding standard gives a uniform appearance to the codes written by different engineers.
- It improves readability and maintainability of the code and it reduces complexity also.
- It helps in code reuse and helps to detect error easily.
- It promotes sound programming practices and increases efficiency of the programmers.

Some of the coding standards are given below:

Limited use of globals: These rules talk about which types of data that can be declared global and the data that can't be.

Standard headers for different modules: For better understanding and maintenance of the code, the header of different modules should follow some standard format and information. The header format must contain below things that is being used in various companies:

- Name of the module
- Date of module creation
- Author of the module
- Modification history
- Synopsis of the module about what the module does

- Different functions supported in the module along with their input output parameters
- Global variables accessed or modified by the module

Naming conventions for local variables, global variables, constants and functions: Some of the naming conventions are given below:

- Meaningful and understandable variables name helps anyone to understand the reason of using it.
- Local variables should be named using camel case lettering starting with small letter (e.g. local Data) whereas Global variables names should start with a capital letter (e.g. Global Data). Constant names should be formed using capital letters only (e.g. CONSDATA).
- It is better to avoid the use of digits in variable names.
- The names of the function should be written in camel case starting with small letters.
- The name of the function must describe the reason of using the function clearly and briefly.

Indentation: Proper indentation is very important to increase the readability of the code. For making the code readable, programmers should use White spaces properly. Some of the spacing conventions are given below:

- There must be a space after giving a comma between two function arguments.
- Each nested block should be properly indented and spaced.
- Proper Indentation should be there at the beginning and at the end of each block in the program.
- All braces should start from a new line and the code following the end of braces also start from a new line.

Error return values and exception handling conventions: All functions that encountering an error condition should either return a 0 or 1 for simplifying the debugging. On the other hand, Coding guidelines give some general suggestions regarding the coding style that to be followed for the betterment of understandability and readability of the code.

Avoid using a coding style that is too difficult to understand: Code should be easily understandable. The complex code makes maintenance and debugging difficult and expensive.

Avoid using an identifier for multiple purposes: Each variable should be given a

descriptive and meaningful name indicating the reason behind using it. This is not possible if an identifier is used for multiple purposes and thus it can lead to confusion to the reader. Moreover, it leads to more difficulty during future enhancements.

Code should be well documented: The code should be properly commented for understanding easily. Comments regarding the statements increase the understandability of the code.

Length of functions should not be very large: Lengthy functions are very difficult to understand. That's why functions should be small enough to carry out small work and lengthy functions should be broken into small ones for completing small tasks.

Try not to use GOTO statement: GOTO statement makes the program unstructured, thus it reduces the understandability of the program and also debugging becomes difficult.

Advantages of Coding Guidelines

- Coding guidelines increase the efficiency of the software and reduces the development time.
- Coding guidelines help in detecting errors in the early phases, so it helps to reduce the extra cost incurred by the software project.
- If coding guidelines are maintained properly, then the software code increases readability and understandability thus it reduces the complexity of the code.
- It reduces the hidden cost for developing the software.

Code Review

Code review is a software quality assurance process in which software's source code is analyzed manually by a team or by using an automated code review tool. The motive is purely, to find bugs, resolve errors, and for most times, improving code quality. Reviewing the codebase makes sure that every software or new feature developed within the company is of high quality. Code review is an essential process that every software company must follow, so we researched the best practices for reviewing code. Code review is a systematic examination of software source code, intended to find bugs and to estimate the code quality.

Code Review Techniques

1. Code Walkthrough: Code Walkthrough is an informal code analysis technique. It is a form of peer review in which a programmer leads the review process and the other team members ask questions and spot possible errors against development standards

and other issues.

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

2. Code Inspection

Code Inspection is the most formal type of review, which is a kind of static testing to avoid the defect multiplication at a later stage.

- The main purpose of code inspection is to find defects and it can also spot any process improvement if any.
- An inspection report lists the findings, which include metrics that can be used to aid improvements to the process as well as correcting defects in the document under review.
- Preparation before the meeting is essential, which includes reading of any source documents to ensure consistency.
- Inspections are often led by a trained moderator, who is not the author of the code.
- The inspection process is the most formal type of review based on rules and checklists and makes use of entry and exit criteria.
- It usually involves peer examination of the code and each one has a defined set of roles.
- After the meeting, a formal follow-up process is used to ensure that corrective action is completed in a timely manner.

3. Clean Room Testing

Clean room testing was pioneered by IBM. this kind of testing depends heavily on walkthroughs, inspection, and formal verification. The programmers don't seem to be allowed to check any of their code by corporal punishment the code apart from doing a

little syntax testing employing a compiler. The computer code development philosophy relies on avoiding computer code defects by employing a rigorous examination method. the target of this computer code is that the zero-defect computer code.

Software Documentation

Software documentation is a manual-cum-guide that helps in understanding and correctly utilizing the software code. The coding standards and naming conventions written in a commonly spoken language in code documentation provide enhanced clarity for the designer. Moreover, they act as a guide for the software maintenance team (this team focuses on maintaining software by improving and enhancing the software after it has been delivered to the end user) while the software maintenance process is carried out. In this way, code documentation facilitates code reusability.

Internal Documentation

Generally, internal documentation comprises the following information:

- Name, type, and purpose of each variable and data structure used in the code
- Brief description of algorithms, logic, and error-handling techniques
- Information about the required input and expected output of the program
- Assistance on how to test the software
- Information on the upgradations and enhancements in the program.

Documentation which focuses on general description of the software code and is not concerned with its detail is known as external documentation. It includes information such as function of code, name of the software developer who has written the code, algorithms used in the software code, dependency of code on programs and libraries, and format of the output produced by the software code. Generally, external documentation includes structure charts for providing an outline of the program and describing the design of the program. Internal documentation is the code comprehension features provided in the source code itself. It can be provided in the code in several forms. The important types of internal documentation are the following:

- Comments embedded in the source code.
- Use of meaningful variable names.
- Module and function headers.
- Code indentation.
- Code structuring.

- Use of enumerated types.
- Use of constant identifiers.
- Use of user-defined data types.

External Documentation

Documentation is useful for software developers as it consists of information such as description of the problem along with the program written to solve it. In addition, it describes the approach used to solve the problem, operational requirements of the program, and user interface components. For the purpose of readability and proper understanding, the detailed description is accompanied by figures and illustrations that show how one component is related to another. External documentation explains why a particular solution is chosen and implemented in the software. It also includes formulas, conditions, and references from where the algorithms or documentation are derived. External documentation makes the user aware of the errors that occur while running the software code.

Testing

Software testing is a process used to identify the correctness, completeness and quality of developed computer software. It is the process of executing a program/ application under positive and negative conditions by manual or automated means. It checks for the specification, functionality and performance. Testing is the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not. In simple words, testing is executing a system in order to identify any gaps, errors, or missing requirements in contrary to the actual requirements.

Why Software Testing?

Software testing is important as it may cause mission failure, impact on operational performance and reliability if not done properly. Effective software testing delivers quality software products satisfying user's requirements, needs and expectations.

Who is Software Tester?

Software tester is the one who performs testing and find bugs, if they exist in the tested application.

Verification

Verification is the process confirming that software meets its specification, done through inspections and walkthroughs. Use to identify defects in the product early in the life cycle.

Validation

Validation is the process confirming that it meets the user's requirements. It is the actual testing.

Verification: is the product right.

Validation: is it the right product.

Verification & Validation

These two terms are very confusing for most people, who use them interchangeably. The following table highlights the differences between verification and validation.

Verification	Validation
Verification addresses the concern: "Are you building it right?"	Validation addresses the concern: "Are you building the right thing?"
Ensures that the software system meets all the functionality.	Ensures that the functionalities meet the intended behavior.
Verification takes place first and includes the checking for documentation, code, etc.	Validation occurs after verification and mainly involves the checking of the overall product.
Done by developers.	Done by testers.
It has static activities, as it includes collecting reviews, walkthroughs, and inspections to verify a software.	It has dynamic activities, as it includes executing the software against the requirements.
It is an objective process and no subjective decision should be needed to verify a software.	It is a subjective process and involves subjective decisions on how well a software works.

Test Characteristics

The following are the good characteristics of test

- A good test has a high probability of finding an error.
- A good test is not redundant.
- A good test should be best of breed.
- A good test should be neither too simple nor too complex.

Black-Box Testing

The technique of testing without having any knowledge of the interior workings of the application is called black-box testing. The tester is oblivious to the system architecture and does not have access to the source code. Typically, while performing a black-box test, a tester will interact with the system's user interface by providing inputs and examining outputs without knowing how and where the inputs are worked upon. The following table lists the advantages and disadvantages of black-box testing. Black-box testing attempts to find errors in the following categories:

- In corrector missing functions
- Interface errors
- Errors in data structures or external database access
- Behavior or performance errors
- Initialization and termination errors.

Advantages

- Well suited and efficient for large code segments.
- Code access is not required.
- Clearly separates user's perspective from the developer's perspective through visibly defined roles.
- Large numbers of moderately skilled testers can test the application with no knowledge of implementation, programming language, or operating systems.

Disadvantages

- Limited coverage, since only a selected number of test scenarios is actually performed.
- Inefficient testing, due to the fact that the tester only has limited knowledge about an application.
- Blind coverage, since the tester cannot target specific code segments or error-prone areas.
- The test cases are difficult to design.

White-Box Testing

White-box testing is the detailed investigation of internal logic and structure of the code. White-box testing is also called glass testing or open-box testing. In order to perform white-box testing on an application, a tester needs to know the internal workings of the

code. The tester needs to have a look inside the source code and find out which unit/chunk of the code is behaving inappropriately.

Advantages

- As the tester has knowledge of the source code, it becomes very easy to find out which type of data can help in testing the application effectively.
- It helps in optimizing the code.
- Extra lines of code can be removed which can bring in hidden defects.
- Due to the tester's knowledge about the code, maximum coverage is attained during test scenario writing.

Disadvantages

- Due to the fact that a skilled tester is needed to perform white-box testing, the costs are increased.
- Sometimes it is impossible to look into every nook and corner to find out hidden errors that may create problems, as many paths will go untested.
- It is difficult to maintain white-box testing, as it requires specialized tools like code analysers and debugging tools.

Difference between Black Box Testing and White Box Testing

Black Box Testing	White Box Testing
In this testing knowledge of programming is not necessarily essential.	In this form of testing knowledge of programming is must means it is essential.
Normally independent software testers are responsible for doing this type of testing.	Normally software developers are responsible for doing this type of testing.
In this form of testing Knowledge of implementation is not required.	In this form of testing Implementation knowledge is required.
In this testing testers may or may not be technically sound.	Normally software developers are doing the white box testing but if it is performed by software testers than testers should be technically sound.
In this sort of testing testers mainly focuses on the functionality of the system.	In this sort of testing developers mainly focuses on the structure means program/code of the system.

This testing is done by testers.	This testing is mostly done by developers.
This type of testing always focuses on what is performing/ carried out.	This type of testing always focuses on how it is performing/ carried out.
In Black Box Testing no knowledge regarding internal logic of code is needed means no need of programming is necessary.	In White Box Testing knowledge regarding internal logic of code is needed means need of programming is mandatory.
Other names of this testing include means synonyms of black box testing are testing regarding functionality means Functional testing, Behavioral testing, and Opaque-box/ Closed-box testing that is the reason why in this testing no knowledge of programming is needed.	Other names of this testing include means synonyms of white box testing are testing regarding code means Structural testing, Glass-box/ Clear-box testing, Open-box testing/ Transparent-box testing, Logic-driven testing and Path-oriented testing that is the reason why in this testing knowledge of programming is needed.
Black box testing means functional test or external test.	White box testing means structural test or interior test.

Debugging

In the context of software engineering, debugging is the process of fixing a bug in the software. In other words, it refers to identifying, analyzing and removing errors. This activity begins after the software fails to execute properly and concludes by solving the problem and successfully testing the software. It is considered to be an extremely complex and tedious task because errors need to be resolved at all stages of debugging. Debugging is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system.

Need for Debugging

Once errors are known during a program code, it's necessary to initial establish the precise program statements liable for the errors and so to repair them.

Debugging Process

Debugging is not testing but often occurs as a consequence of testing. The debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many

cases, the non-corresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction. Steps involved in debugging are:

- Problem identification and report preparation.
- Assigning the report to software engineer to the defect to verify that it is genuine.
- Defect Analysis using modeling, documentations, finding and testing candidate flaws, etc.
- Defect Resolution by making required changes to the system.
- Validation of corrections.

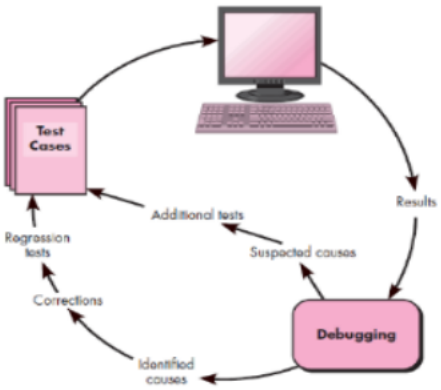


Fig 17.7: The debugging process

There are four steps involved in debugging process

- ✔ Identify
- ✔ Isolate
- ✔ Fix
- ✔ Review

Debugging Approaches

The following are a number of approaches popularly adopted by programmers for debugging

1. Brute Force Method

This is the foremost common technique of debugging however is that the least economical method. during this approach, the program is loaded with print statements to print the intermediate values with the hope that a number of the written values can

facilitate to spot the statement in error. This approach becomes a lot of systematic with the utilization of a symbolic program (also known as a source code debugger), as a result of values of various variables will be simply checked and breakpoints and watch-points can be easily set to check the values of variables effortlessly.

2. Backtracking

This is additionally a reasonably common approach. during this approach, starting from the statement at which an error symptom has been discovered, the source code is derived backward till the error is discovered. sadly, because the variety of supply lines to be derived back will increase, the quantity of potential backward methods will increase and should become unimaginably large so limiting the utilization of this approach.

3. Cause Elimination Method

In this approach, a listing of causes that may presumably have contributed to the error symptom is developed and tests are conducted to eliminate every error. A connected technique of identification of the error from the error symptom is that the package fault tree analysis.

4. Program Slicing

This technique is analogous to backtracking. Here the search house is reduced by process slices. A slice of a program for a specific variable at a particular statement is that the set of supply lines preceding this statement which will influence the worth of that variable

Debugging Guidelines

Debugging is commonly administrated by programmers supported their ingenuity. The subsequent are some general tips for effective debugging:

- Many times, debugging needs an intensive understanding of the program style.
- Debugging might generally even need a full plan of the system.
- One should be watched out for the likelihood that a slip correction might introduce new errors.

Integration Testing

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design. Integration testing is a process of testing the interface between two software units or modules. It's focus on determining the correctness of

the interface. The purpose of the integration testing to expose faults int the interaction between integrated units. Once all the modules have been unit tested, integration testing is performed.

Integration test Approaches

The following are the integration testing approaches:

Big-Bang Integration Testing

It is the simplest integration testing approach, where all the modules are combining and verifying the functionality after the completion of individual module testing. In simple words, all the modules of the system are simply put together and tested. This approach is practicable only for very small systems. If once an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated. So, debugging errors reported during big bang integration testing are very expensive to fix.

Mixed Integration Testing

A mixed integration testing is also called sandwiched integration testing. A mixed integration testing follows a combination of top down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level module have been coded and unit tested. In bottom-up approach, testing can start only after the bottom level modules are ready. This sandwich or mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. A mixed integration testing is also called sandwiched integration testing.

Top-Down Integration Testing

Top-down integration testing technique used in order to simulate the behavior of the lower-level modules that are not yet integrated. In this testing takes place from top to bottom. First high-level modules are tested and then low-level modules and finally integrating the low-level modules to a high level to ensure the system is working as intended.

There are two ways to integrates integration testing

1. Depth First Integration
2. Breadth First Integration

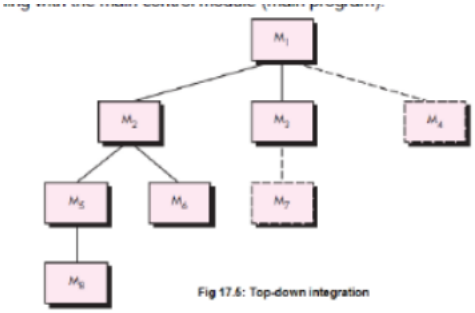


Fig 17.5: Top-down integration

Bottom-Up Integration

Bottom-up integration testing, each module at lower levels is tested with higher modules until all modules are tested. The primary purpose of this integration testing is each subsystem is to test the interfaces among various modules making up the subsystem. This integration testing uses test drivers to drive and pass appropriate data to the lower-level modules. A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters that perform a specific software sub-function.
2. A driver is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

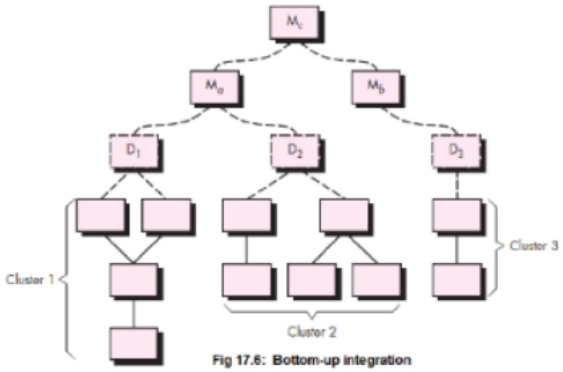


Fig 17.6: Bottom-up integration

There are two ways to test bottom-up integration testing

1. Regression Testing
2. Smoke Testing

Regression Testing: Regression testing helps to ensure that changes do not introduce unintended behavior or additional errors. Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools. The regression test suite contains three different classes of test cases:

• A representative sample of tests that will exercise all software functions.

• Additional tests that focus on software functions that are likely to be affected by the change.

• Tests that focus on the software components that have been changed.

Smoke Testing: Smoke testing is an integration testing approach that is commonly used when product software is developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis. In essence, the smoke testing approach encompasses the following activities:

1. Software components that have been translated into code are integrated into a build.

2. A series of tests is designed to expose errors that will keep the build from properly performing its function.

3. The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily.

Smoke testing provides a number of benefits when it is applied on complex, time critical, software projects:

• Integration risk is minimized.

• The quality of the end product is improved.

• Error diagnosis and correction are simplified.

• Progress is easier to assess.

Program Analysis Tools

Program Analysis Tool is an automated tool whose input is the source code or the executable code of a program and the output is the observation of characteristics of the program. It gives various characteristics of the program such as its size, complexity, adequacy of commenting, adherence to programming standards and many other characteristics.

Classification of Program Analysis Tools

Program Analysis Tools are classified into two categories:

1. Static Program Analysis Tools

Static Program Analysis Tool is such a program analysis tool that evaluates and computes various characteristics of a software product without executing it. Normally, static program analysis tools analyze some structural representation of a program to reach a certain analytical conclusion. Basically, some structural properties are analyzed using static program analysis tools. The structural properties that are usually analyzed are:

- Whether the coding standards have been fulfilled or not.
- Some programming errors such as uninitialized variables.
- Mismatch between actual and formal parameters.
- Variables that are declared but never used.

2. Dynamic Program Analysis Tools

Dynamic Program Analysis Tool is such type of program analysis tool that require the program to be executed and its actual behavior to be observed. A dynamic program analyzer basically implements the code. It adds additional statements in the source code to collect the traces of program execution. When the code is executed, it allows us to observe the behavior of the software for different test cases. Once the software is tested and its behavior is observed, the dynamic program analysis tool performs a post execution analysis and produces reports which describe the structural coverage that has been achieved by the complete testing process for the program.

System Testing

System testing is a large computer-based testing, actually a series of different tests whose primary purpose is to fully exercise the computer-based system. System testing is defined as testing of a complete and full integrated software product. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

Types of System Testing

1 Recovery Testing: Many computer-based systems must recover from faults and resume processing with little or no downtime. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur. Recovery testing is a system test that forces the

software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), re-initialization, check pointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

2 Security Testing: Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport, disgruntled employees who attempt to penetrate for revenge, dishonest individuals who attempt to penetrate for illicit personal gain.

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes! The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to break down any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry. Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

3 Stress Testing: Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example, (1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate, (2) input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other resources are executed, (4) test cases that may cause thrashing in a virtual operating system are designed, (5) test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

A variation of stress testing is a technique called sensitivity testing. In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even

erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

4 Performance Testing: Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained. Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion.

5 Deployment Testing: In many cases, software must execute on a variety of platforms and under more than one operating system environment. Deployment testing, sometimes called configuration testing, exercises the software in each environment in which it is to operate. In addition, deployment testing examines all installation procedures and specialized installation software (e.g., “installers”) that will be used by customers, and all documentation that will be used to introduce the software to end users.

Performance Testing

Performance Testing is a type of software testing that ensures software applications to perform properly under their expected workload. It is a testing technique carried out to determine system performance in terms of sensitivity, reactivity and stability under a particular workload. Performance Testing is the process of analyzing the quality and capability of a product. It is a testing method performed to determine the system performance in terms of speed, reliability and stability under varying workload. Performance testing is also known as Perf Testing.

Performance Testing Attributes

Speed: It determines whether the software product responds rapidly.

Scalability: It determines amount of load the software product can handle at a time.

Stability: It determines whether the software product is stable in case of varying workloads.

Reliability: It determines whether the software product is secure or not.

Objective of Performance Testing

- The objective of performance testing is to eliminate performance congestion.
- It uncovers what is needed to be improved before the product is launched in market.
- The objective of performance testing is to make software rapid.
- The objective of performance testing is to make software stable and reliable.

Regression Testing

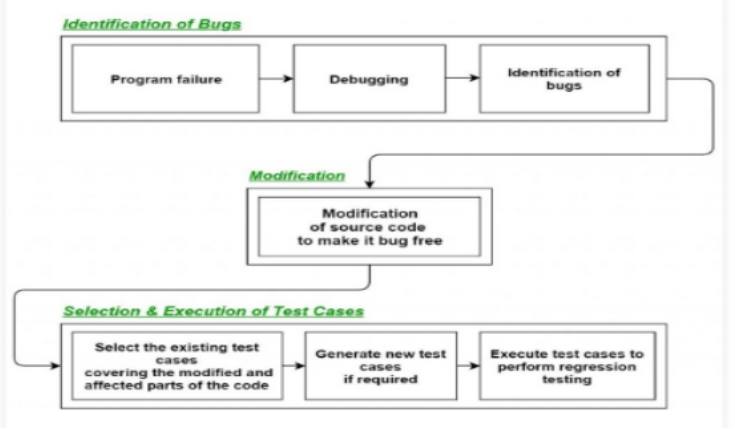
Regression Testing is the process of testing the modified parts of the code and the parts that might get affected due to the modifications to ensure that no new errors have been introduced in the software after the modifications have been made. Regression means return of something and in the software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

When to do regression testing?

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

Process of Regression testing

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



When can we perform Regression Testing?

- We do regression testing whenever the production code is modified.
- We can perform regression testing in the following scenario, these are:
  - When new functionality added to the application.
  - When there is a Change Requirement.
  - When the defect fixed.
  - When there is a performance issue fix.
  - When there is an environment change.

Testing Object-Oriented Programs

Whenever large-scale systems are designed, object-oriented testing is done rather than the conventional testing strategies as the concepts of object-oriented programming is way different from that of conventional ones. The whole object-oriented testing revolves around the fundamental entity known as “class”. With the help of “class” concept, larger systems can be divided into small well-defined units which may then be implemented separately.

The object-oriented testing can be classified as like conventional systems. These are called as the levels for testing. The levels of object-oriented testing can be broadly classified into three categories. These are:

Class Testing

Class testing is also known as unit testing. In class testing, all individual classes are tested for errors or bugs. Class testing ensures that the attributes of class are

implemented as per the design and specifications. Also, it checks whether the interfaces and methods are error free of not.

Inter-Class Testing

It is also called as integration or subsystem testing. Inter class testing involves the testing of modules or sub-systems and their coordination with other modules.

System Testing

In system testing, the system is tested as whole and primarily functional testing techniques are used to test the system. Non-functional requirements like performance, reliability, usability and test-ability are also tested

In object-oriented programs, control flow is characterized by message passing among objects, and the control flow switches from one object to another by inter-object communication. Consequently, there is no control flow within a class like functions. This lack of sequential control flow within a class requires different approaches for testing. Furthermore, in a function, arguments passed to the function with global data determine the path of execution within the procedure. But, in an object, the state associated with the object also influences the path of execution, and methods of a class can communicate among themselves through this state because this state is persistent across invocations of methods. Hence, for testing objects, the state of an object has to play an important role.

Techniques of object-oriented testing

The following are the techniques of object-oriented testing:

Fault Based Testing:

This type of checking permits for coming up with test cases supported the consumer specification or the code or both. It tries to identify possible faults (areas of design or code that may lead to errors.). For all of these faults, a test case is developed to “flush” the errors out. These tests also force each time of code to be executed. This method of testing does not find all types of errors. However, incorrect specification and interface errors can be missed. These types of errors can be uncovered by function testing in the traditional testing model. In the object-oriented model, interaction errors can be uncovered by scenario-based testing. This form of Object oriented-testing can only test against the client’s specifications, so interface errors are still missed.

Class Testing Based on Method Testing:

This approach is the simplest approach to test classes. Each method of the class

performs a well- defined cohesive function and can, therefore, be related to unit testing of the traditional testing techniques. Therefore, all the methods of a class can be involved at least once to test the class.

Random Testing:

It is supported by developing a random test sequence that tries the minimum variety of operations typical to the behavior of the categories

Partition Testing:

This methodology categorizes the inputs and outputs of a category so as to check them severely. This minimizes the number of cases that have to be designed.

Scenario-based Testing:

It primarily involves capturing the user actions then stimulating them to similar actions throughout the test. These tests tend to search out interaction form of error.

UNIT-5

Software Reliability

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

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

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

What is Statistical Testing (ST)?

Statistical Testing makes use of statistical methods to determine the reliability of the program. Statistical testing focuses on how faulty programs can affect its operating conditions.

How to perform ST?

- Software is tested with the test data that statistically models the working environment.
- Failures are collated and analyzed.
- From the computed data, an estimate of program's failure rate is calculated.
- A Statistical method for testing the possible paths is computed by building an algebraic function.
- Statistical testing is a bootless activity as the intent is NOT to find defects.

Software Quality and Management

Software Quality

Software quality product is defined in term of its fitness of purpose. That is, a quality product does precisely what the users want it to do. For software products, the fitness of use is generally explained in terms of satisfaction of the requirements laid down in the SRS document.

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

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

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

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

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

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

Software Quality Management System

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

A quality system subsists of the following:

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

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

Auditing of projects

Review of the quality system

Development of standards, methods, and guidelines, etc.

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

ISO 9000

ISO (International Standards Organization) is a group or consortium of 63 countries established to plan and fosters standardization. ISO declared its 9000 series of standards in 1987. It serves as a reference for the contract between independent parties. The ISO 9000 standard determines the guidelines for maintaining a quality system. The ISO standard mainly addresses operational methods and organizational methods such as responsibilities, reporting, etc. ISO 9000 defines a set of guidelines for the production process and is not directly concerned about the product itself.

Types of ISO 9000 Quality Standards

ISO 9001: This standard applies to the organizations engaged in design, development, production, and servicing of goods. This is the standard that applies to most software development organizations.

ISO 9002: This standard applies to those organizations which do not design products but are only involved in the production. Examples of these category industries contain steel and car manufacturing industries that buy the product and plants designs from external sources and are engaged in only manufacturing those products. Therefore, ISO 9002 does not apply to software development organizations.

ISO 9003: This standard applies to organizations that are involved only in the installation and testing of the products. For example, Gas companies.

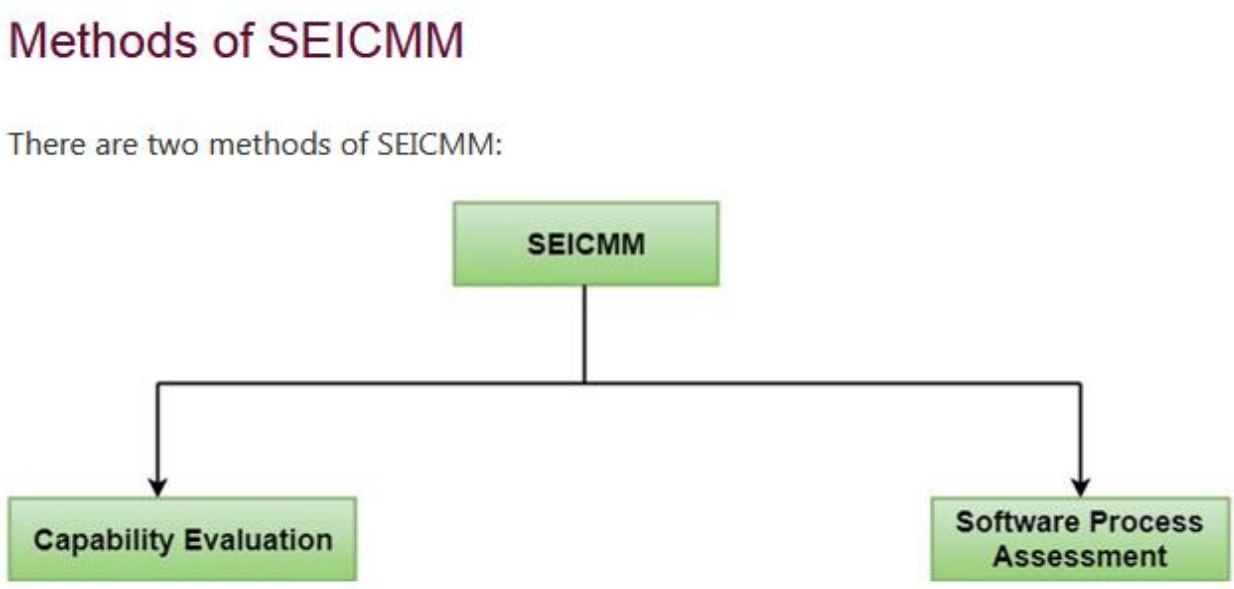
Software Engineering Institute Capability Maturity Model (SEICMM)

The Capability Maturity Model (CMM) is a procedure used to develop and refine an organization's software development process.

The model defines a five-level evolutionary stage of increasingly organized and consistently more mature processes.

CMM was developed and is promoted by the Software Engineering Institute (SEI), a research and development center promote by the U.S. Department of Defense (DOD).

Capability Maturity Model is used as a benchmark to measure the maturity of an organization's software process.

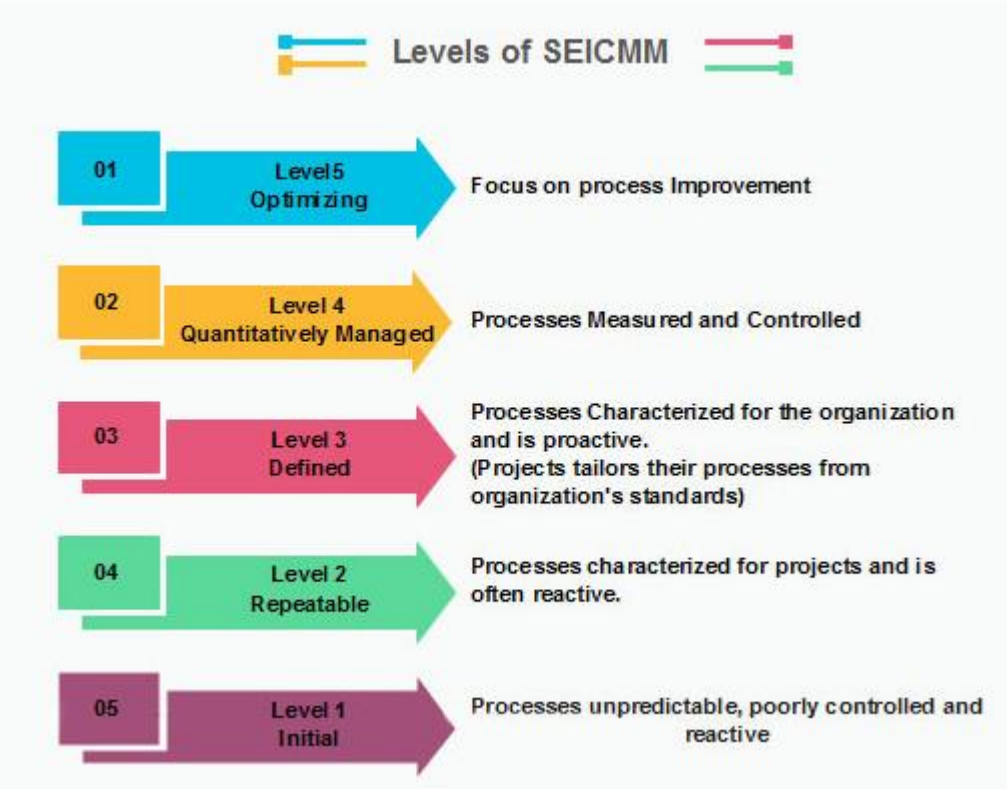


Capability Evaluation: Capability evaluation provides a way to assess the software process capability of an organization. The results of capability evaluation indicate the likely contractor

performance if the contractor is awarded a work. Therefore, the results of the software process capability assessment can be used to select a contractor.

Software Process Assessment: Software process assessment is used by an organization to improve its process capability. Thus, this type of evaluation is for purely internal use.

SEI CMM categorized software development industries into the following **five** maturity levels.



Level 1: Initial

Ad hoc activities characterize a software development organization at this level. Very few or no processes are described and followed. Since software production processes are not limited, different engineers follow their process and as a result, development efforts become chaotic. Therefore, it is also called a chaotic level.

Level 2: Repeatable

At this level, the fundamental project management practices like tracking cost and schedule are established. Size and cost estimation methods, like function point analysis, COCOMO, etc. are used.

Level 3: Defined

At this level, the methods for both management and development activities are defined and documented. There is a common organization-wide understanding of operations, roles, and

responsibilities. The ways through defined, the process and product qualities are not measured. ISO 9000 goals at achieving this level.

Level 4: Managed

At this level, the focus is on software metrics. **Two kinds of metrics** are composed.

Product metrics measure the features of the product being developed, such as its size, reliability, time complexity, understandability, etc.

Process metrics follow the effectiveness of the process being used, such as average defect correction time, productivity, the average number of defects found per hour inspection, the average number of failures detected during testing per LOC, etc.

Level 5: Optimizing

At this phase, process and product metrics are collected. Process and product measurement data are evaluated for continuous process improvement.

Personal Software Process (PSP)

The ***Personal Software Process*** **(PSP)** emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product. In addition **PSP** makes the practitioner responsible for project planning and empowers the practitioner to control the quality of all software work products that are developed. The **PSP** model defines **five** framework activities:

- ✓ **Planning.** This activity isolates requirements and develops both size and resource estimates. In addition, defects estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.
- ✓ **High-level design.** External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.
- ✓ **High-level design review.** Formal verification methods are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

- ✓ **Development.** The component-level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results.
- ✓ **Postmortem.** Using the measures and metrics collected, the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

PSP stresses the need to identify errors early and, just as important, to understand the types of errors that you are likely to make.

Six Sigma

Six Sigma is a highly disciplined process that helps us focus on developing and delivering near-perfect products and services.

Features of Six Sigma

- Six Sigma's aim is to eliminate waste and inefficiency, thereby increasing customer satisfaction by delivering what the customer is expecting.
- Six Sigma follows a structured methodology, and has defined roles for the participants.
- Six Sigma is a data driven methodology, and requires accurate data collection for the processes being analyzed.

- Six Sigma is about putting results on Financial Statements.
- Six Sigma is a business-driven, multi-dimensional structured approach for −
    - Improving Processes
    - Lowering Defects
    - Reducing process variability
    - Reducing costs
    - Increasing customer satisfaction
    - Increased profits

The word *Sigma* is a statistical term that measures how far a given process deviates from perfection.

The central idea behind Six Sigma: If you can measure how many "defects" you have in a process, you can systematically figure out how to eliminate them and get as close to "zero defects" as possible and specifically it means a failure rate of 3.4 parts per million or 99.9997% perfect.

Key Concepts of Six Sigma

At its core, Six Sigma revolves around a few key concepts.

- **Critical to Quality** − Attributes most important to the customer.
- **Defect** − Failing to deliver what the customer wants.
- **Process Capability** − What your process can deliver.
- **Variation** − What the customer sees and feels.
- **Stable Operations** − Ensuring consistent, predictable processes to improve what the customer sees and feels.
- **Design for Six Sigma** − Designing to meet customer needs and process capability.

Our Customers Feel the Variance, Not the Mean. So Six Sigma focuses first on reducing process variation and then on improving the process capability.

CASE and its Scope

A CASE (Computer power-assisted software package Engineering) tool could be a generic term accustomed denote any type of machine-driven support for software package engineering. In a very additional restrictive sense, a CASE tool suggests that any tool accustomed automatize some activity related to software package development.

Several CASE tools square measure obtainable. A number of these CASE tools assist in part connected tasks like specification, structured analysis, design, coding, testing, etc.; and other to non-phase activities like project management and configuration management.

Reasons for using CASE tools:

The primary reasons for employing a CASE tool are:

- To extend productivity
- To assist turn out higher quality code at a lower price

CASE Environment

Although individual CASE tools square measure helpful, the true power of a toolset is often completed only this set of tools square measure integrated into a typical framework or setting. CASE tools square measure characterized by the stage or stages of package development life cycle that they focus on. Since totally different tools covering different stages share common data, it's needed that they integrate through some central repository to possess an even read of data related to the package development artifacts. This central repository is sometimes information lexicon containing the definition of all composite and elementary data things.

- Through the central repository, all the CASE tools in a very CASE setting share common data among themselves. therefore a CASE setting facilities the automation of the step-wise methodologies for package development. A schematic illustration of a CASE setting is shown in the below diagram:

A CASE environment facilitates the automation of the in small stages methodologies for package development. In distinction to a CASE environment, a programming environment is an Associate in a Nursing integrated assortment of tools to support solely the cryptography part of package development.

Software Maintenance

Software maintenance is a part of the Software Development Life Cycle. Its primary goal is to modify and update software application after delivery to correct errors and to improve performance. Software is a model of the real world. When the real world changes, the software require alteration wherever possible.

Software Maintenance is an inclusive activity that includes error corrections, enhancement of capabilities, deletion of obsolete capabilities, and optimization.

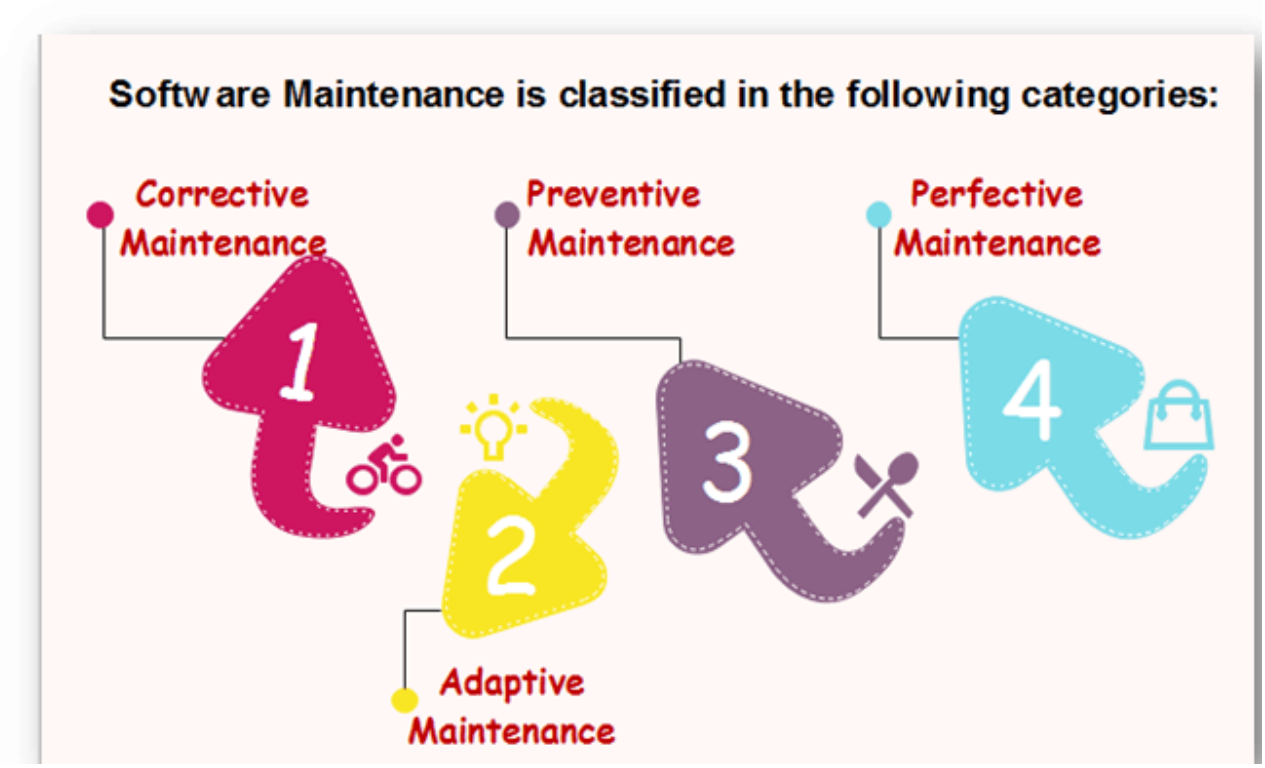
Need for Maintenance

Software Maintenance is needed for:-

- ✓ Correct errors
- ✓ Change in user requirement with time
- ✓ Changing hardware/software requirements
- ✓ To improve system efficiency
- ✓ To optimize the code to run faster
- ✓ To modify the components
- ✓ To reduce any unwanted side effects.

Thus the maintenance is required to ensure that the system continues to satisfy user requirements.

Types of Software Maintenance



1. Corrective Maintenance

Corrective maintenance aims to correct any remaining errors regardless of where they may cause specifications, design, coding, testing, and documentation, etc.

2. Adaptive Maintenance

It contains modifying the software to match changes in the ever-changing environment.

3. Preventive Maintenance

It is the process by which we prevent our system from being obsolete. It involves the concept of reengineering & reverse engineering in which an old system with old technology is re-engineered using new technology. This maintenance prevents the system from dying out.

4. Perfective Maintenance

It defines improving processing efficiency or performance or restricting the software to enhance changeability. This may contain enhancement of existing system functionality, improvement in computational efficiency, etc.

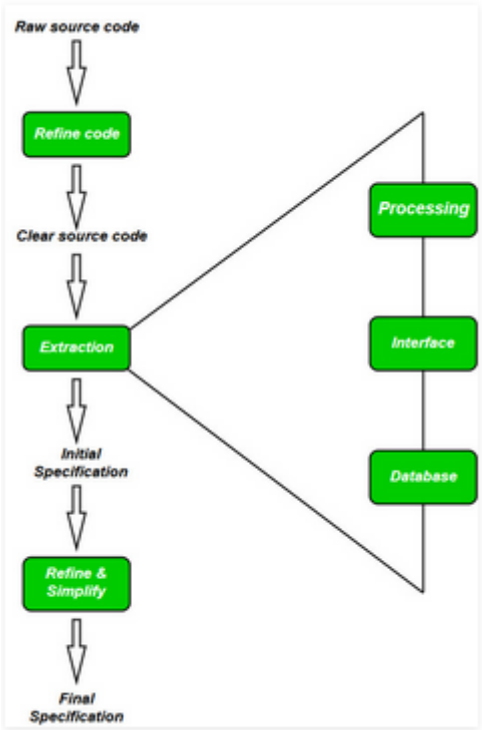
Software Reverse Engineering

Software Reverse Engineering is a process of recovering the design, requirement specifications and functions of a product from an analysis of its code. It builds a program database and generates information from this.

The purpose of reverse engineering is to facilitate the maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system.

Reverse Engineering Goals:

- ✓ Cope with Complexity.
- ✓ Recover lost information.
- ✓ Detect side effects.
- ✓ Synthesise higher abstraction.
- ✓ Facilitate Reuse.



Steps of Software Reverse Engineering:

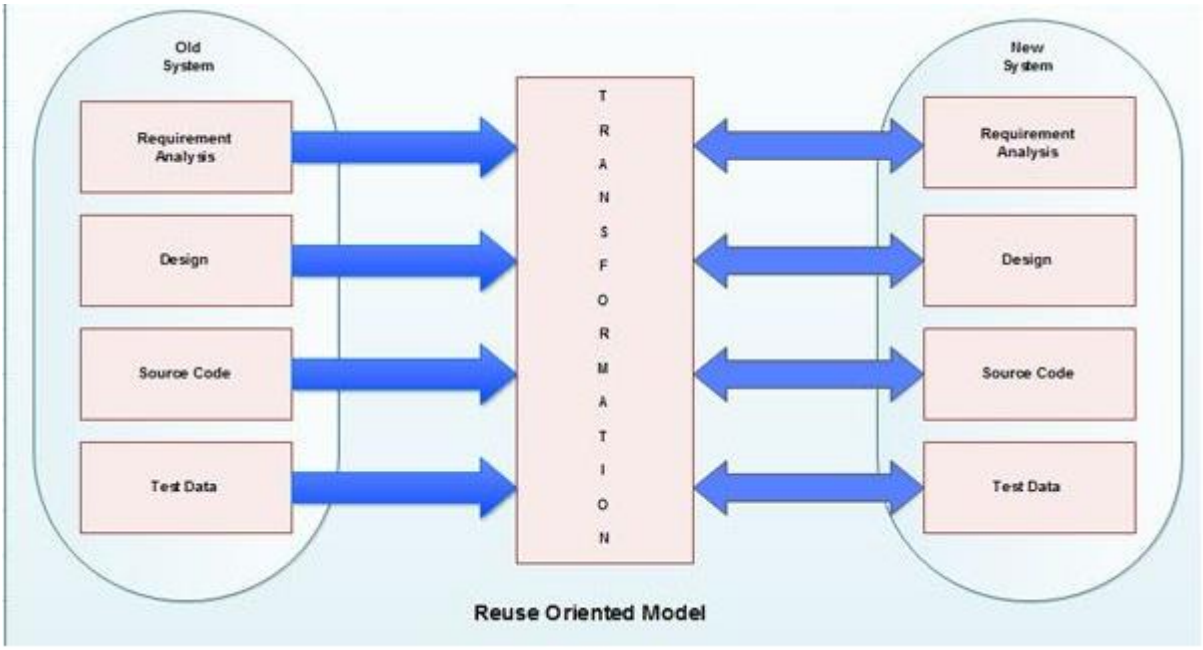
1. **Collection Information:**
   This step focuses on collecting all possible information (i.e., source design documents etc.) about the software.
2. **Examining the information:**
   The information collected in step-1 as studied so as to get familiar with the system.

3. **Extracting the structure:**
   This step concerns with identification of program structure in the form of structure chart where each node corresponds to some routine.
4. **Recording the functionality:**
   During this step processing details of each module of the structure, charts are recorded using structured language like decision table, etc.
5. **Recording data flow:**
   From the information exA tracted in step-3 and step-4, set of data flow diagrams are derived to show the flow of data among the processes.
6. **Recording control flow:**
   High level control structure of the software is recorded.
7. **Review extracted design:**
   Design document extracted is reviewed several times to ensure consistency and correctness. It also ensures that the design represents the program.
8. **Generate documentation:**
   Finally, in this step, the complete documentation including SRS, design document, history, overview, etc. are recorded for future use.

Software Maintenance Processes Model

The Reuse-oriented Model

The reuse-oriented model assumes that the existing program components can be reused to perform maintenance.



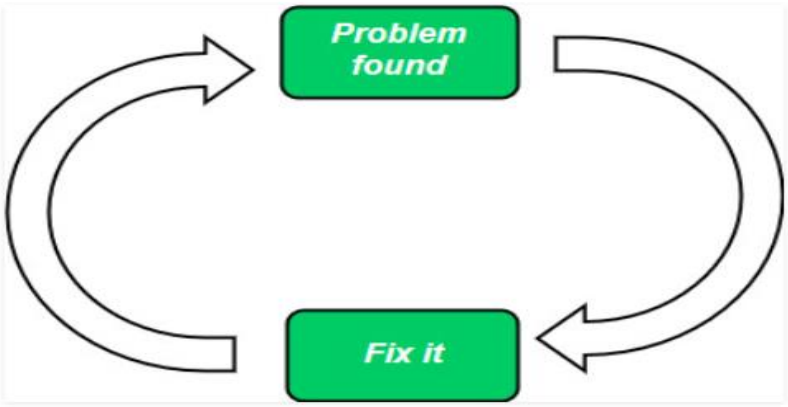
It consists of the following steps.

1. Identifying the components of the old system which can be reused
2. Understanding these components

3. Modifying the old system components so that they can be used in the new system
4. Integrating the modified components into the new system

Quick-fix Model:

- ✓ It is basically an adhoc approach to maintain software.
- ✓ It is a fire fighting approach waiting for the problem to accur and then trying to fix it as quick as possible.
- ✓ The main objective of this model is to identify the problem and then fix it as soon as possible.
- ✓ In this model, changes are made at code level as early as possible without accepting future problems.
- ✓ This model is an approach to modify the software code without little consideration of its impact on the overall structure of the software system.
- ✓ As a result of this model, the structure of the software degrade rapidly



Advantages:

1. The main advantage is that it performs its work at low cost and very quickly.
2. Sometimes, users don't wait for the long time. Rather, they require the modified software to be delivered to them in the least possible time. As a result, the software maintenance team needs to use a quick-fix model to avoid the time consuming process of Software maintenance life cycle.
3. This model is also advantageous in situations where the software system is to be maintained with certain deadlines and limited resources.

Disadvantages:

1. This model is not suitable for large project system.
2. This model is not suitable to fix errors for a longer period, as the structure of the software system degrade rapidly.

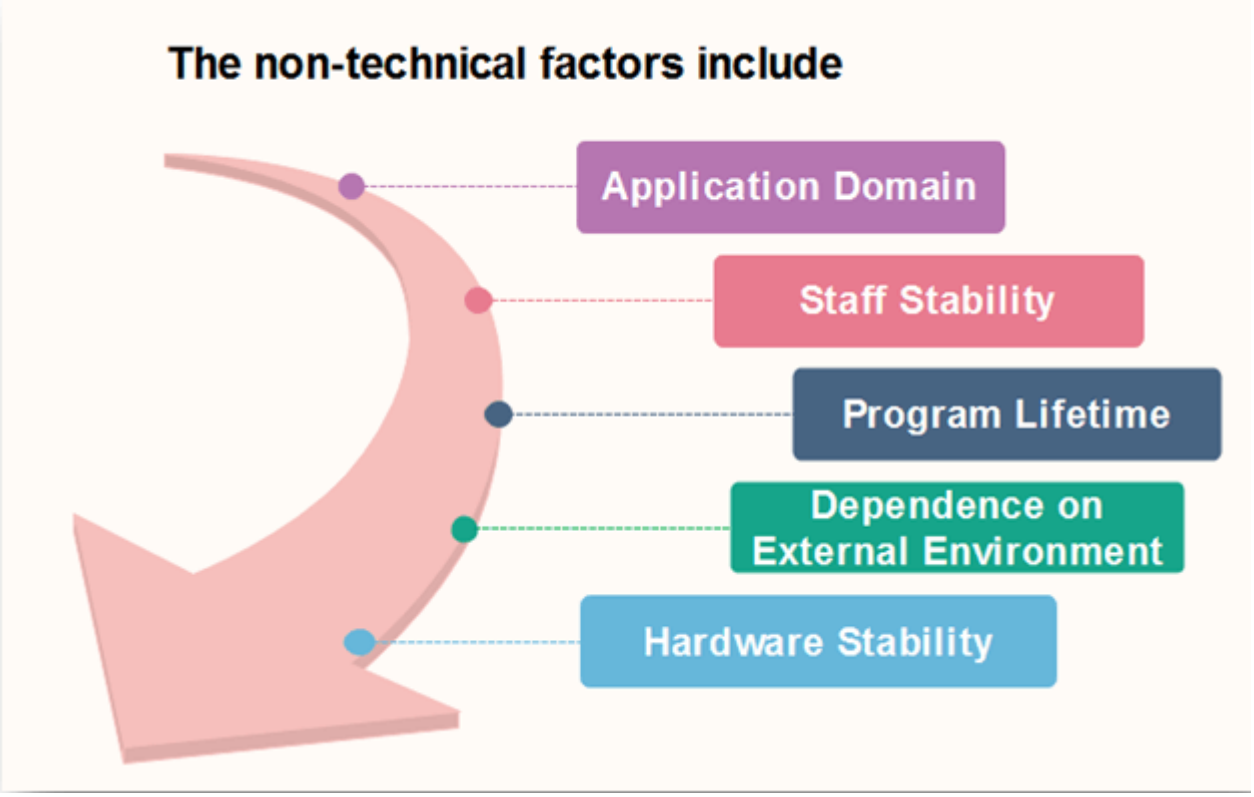
Software Maintenance Cost Factors

There are two types of cost factors involved in software maintenance.

These are

- Non-Technical Factors
- Technical Factors

Non-Technical Factors



1. Application Domain

- If the application of the program is defined and well understood, the system requirements may be definitive and maintenance due to changing needs minimized.
- If the form is entirely new, it is likely that the initial conditions will be modified frequently, as user gain experience with the system.

2. Staff Stability

- ✓ It is simple for the original writer of a program to understand and change an application rather than some other person who must understand the program by the study of the reports and code listing.
- ✓ If the implementation of a system also maintains that systems, maintenance costs will reduce.

3. Program Lifetime

- Programs become obsolete when the program becomes obsolete, or their original hardware is replaced, and conversion costs exceed rewriting costs.

4. Dependence on External Environment

- If an application is dependent on its external environment, it must be modified as the climate changes.

For example:

✓ Changes in a taxation system might need payroll, accounting, and stock control programs to be modified.

✓ Taxation changes are nearly frequent, and maintenance costs for these programs are associated with the frequency of these changes.

5. Hardware Stability

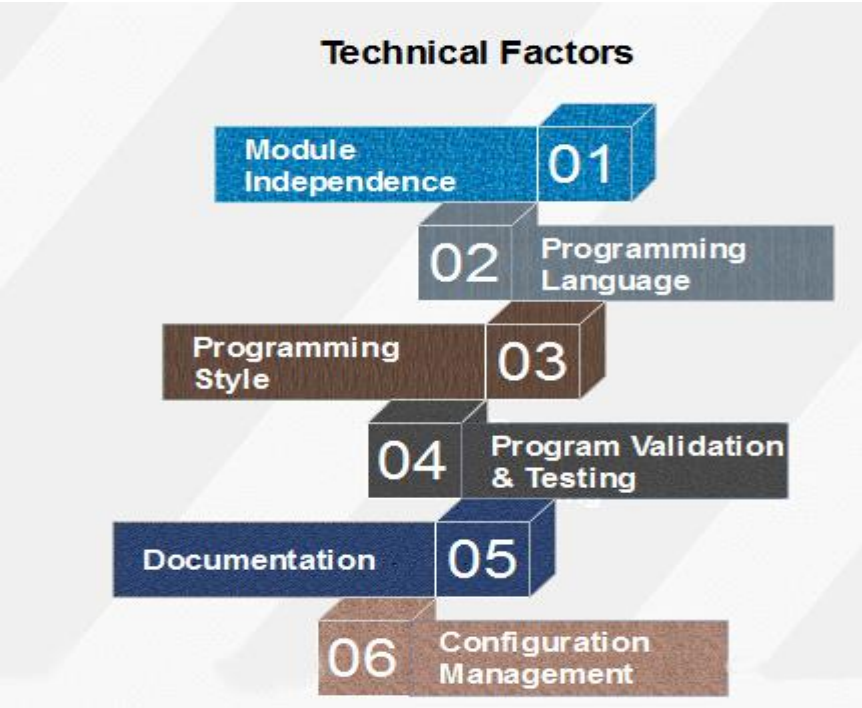
✓ If an application is designed to operate on a specific hardware configuration and that configuration does not changes during the program's lifetime, no maintenance costs due to hardware changes will be incurred.

✓ Hardware developments are so increased that this situation is rare.

✓ The application must be changed to use new hardware that replaces obsolete equipment.

Technical Factors

Technical Factors include the following:



Module Independence

It should be possible to change one program unit of a system without affecting any other unit.

Programming Language

Programs written in a high-level programming language are generally easier to understand than programs written in a low-level language.

Programming Style

The method in which a program is written contributes to its understandability and hence, the ease with which it can be modified.

Program Validation and Testing

- Generally, more the time and effort are spent on design validation and program testing, the fewer bugs in the program and, consequently, maintenance costs resulting from bugs correction are lower.
- Maintenance costs due to bug's correction are governed by the type of fault to be repaired.
- Coding errors are generally relatively cheap to correct, design errors are more expensive as they may include the rewriting of one or more program units.
- Bugs in the software requirements are usually the most expensive to correct because of the drastic design which is generally involved.

Documentation

- If a program is supported by clear, complete yet concise documentation, the functions of understanding the application can be associatively straight-forward.
- Program maintenance costs tends to be less for well-reported systems than for the system supplied with inadequate or incomplete documentation.

Configuration Management Techniques

- One of the essential costs of maintenance is keeping track of all system documents and ensuring that these are kept consistent.
- Effective configuration management can help control these costs.

Basic issues in any reuse program

The following are some of the basic issues that must be clearly understood for starting any reuse program. • Component creation

• Component indexing and storing

• Component search

• Component understanding

• Component adaptation

• Repository maintenance

Component creation– For component creation, the reusable components have to be first identified. Selection of the right kind of components having potential for reuse is important. Domain analysis is a promising technique which can be used to create reusable components.

Component indexing and storing– Indexing requires classification of the reusable components so that they can be easily searched when looking for a component for reuse. The components need to be stored in a Relational Database Management System (RDBMS) or an Object-Oriented Database System (ODBMS) for efficient access when the number of components becomes large.

Component searching– The programmers need to search for right components matching their requirements in a database of components. To be able to search components efficiently, the programmers require a proper method to describe the components that they are looking for.

Component understanding– The programmers need a precise and sufficiently complete understanding of what the component does to be able to decide whether they can reuse the component. To facilitate understanding, the components should be well documented and should do something simple.

Component adaptation– Often, the components may need adaptation before they can be reused, since a selected component may not exactly fit the problem at hand. However, tinkering with the code is also not a satisfactory solution because this is very likely to be a source of bugs.

Repository maintenance– A component repository once is created requires continuous maintenance. New components, as and when created have to be entered into the repository.The faulty components have to be tracked. Further, when new applications emerge, the older applications become obsolete. In this case, the obsolete components might have to be removed from the repository.

Reuse at Organization level

- ✓ Reusability should be a standard part in: Specification, design, implementation, test.
- ✓ Ideally there is a steady flow of reusable components.
- ✓ Extracting reusable knowledge from past projects.
- ✓ Development of new systems leads to assortment of products.
- ✓ Steps for reusable component creation.
  - ✓ Assess product's potential for reuse.
  - ✓ Refine product for greater reusability.
  - ✓ Integrate product with reuse repository.