LEARNING MATERIAL

ON

SOFTWARE ENGINEERING

(FOR 3RD SEMESTER CSE)

PREPARED BY:

PRIYANKA MAHARANA

GOVT.POLYTECHNIC,DHENKANL

CHAPTER-1

SOFTWARE ENGINEERING

Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures.
The outcome of software engineering is an efficient and reliable software product.
Definitions-IEEE defines software engineering as:

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.
(2) The study of approaches as in the above statement.

PROGRAM VBS SOFTWARE PRODUCT program

- They are usually small in size. They are lines of code or maybe 100 to 2000 lines codes on little more.
- There is no documentation or lack in documentation.

Software Product:

- Very big in size. The lines of codes are in thousands To lacs maybe more, depends on Software Product.
- Proper documentation and well documented and user manual prepared. Large or vast
- **Emergence of Software Engineering**
- Software engineering techniques have evolved over many years which resulted series
- of innovations and experience about writing good quality programs.

Early Computer Programming (1950s):

- Programs were being written in assembly language.
- Programs were limited to about a few hundreds of lines of assembly code.
- **High-Level Language Programming (Early 60s)**
- - High-level languages such as FORTRAN, ALGOL, and COBOL were
  - introduced:
  - This reduced software development efforts greatly.
  - **Control Flow-Based Design (late 60s)**
- Programmers found it increasingly difficult not only to write cost effective and correct programs, but also to understand and maintain programs written by others.

Data Structure-Oriented Design

- Software engineers were now expected to develop larger more complicated software products which often required writing in excess of several tens of thousands of lines of source code

Object-Oriented Design

- An object-Oriented design technique is an intuitively appealing approach, where the natural objects occurring in a problem are first identified and then the relationships the objects such as composition, reference, and inheritance are determined

SOFTWARE LIFE CYCLE MODELS

The Software Development Lifecycle is a systematic process for building software that ensures the quality and correctness of the software built.

The software development life cycle (SDLC) is a framework defining tasks performed at each step in the software development process.

Feasibility Study

A feasibility study is simply an assessment of the practicality of a proposed plan or project. As the name implies, these studies ask: Is this project feasible or not .

The goals of feasibility studies are as follows:

1. To understand thoroughly all aspects of a project, concept, or plan
2. To become aware of any potential problems that could occur while implementing the project

Requirement Analysis

During this phase, all the relevant information is collected from the customer to develop a product as per their expectation. Any ambiguities must be resolved in this phase only.

Design

In this phase, the requirement gathered in the SRS document is used as an input and software architecture that is used for implementing system development is derived.

Implementation or Coding

Implementation/Coding starts once the developer gets the Design document. The Software design is translated into source code. All the components of the software are implemented in this phase.

Testing

Testing starts once the coding is complete and the modules are released for testing. In this phase, the developed software is tested thoroughly and any defects found are assigned to developers to get them fixed.

Deployment

Once the product is tested, it is deployed in the production environment or firstUAT(UserAcceptancetesting)is done depending on the customer expectation.

CLASSICAL WATERFALL MODEL AND ITERATIVE WATERFALL MODEL

This model is called as linear sequential model. This model suggests a systematic approach to software development.The project development is divided into sequence of well-defined phases

Different phases of this model are: ·Feasibility study
·Requirements analysis and specification
·Design
·Coding and unit testing
·Integration and system testing
·Maintenance

Feasibility Study

The main of the feasibility study is to determine whether it would be financially, technically and operationally feasible to develop the product. The feasibility study activity involves the analysis of the problem and collection of all relevant information relating to the product such as the different data items which would be input to the system.

- **TechnicalFeasibility**

  Can the work for the project be done with current equipment, existing software technology and available personnel?

- **Economic Feasibility**

  Are there sufficient benefits in creating the system to make the costs acceptable?

- **Operational Feasibility**

  Will the system be used if it is developed and implemented?

Requirement Analysis and Specifications

The goal of this phase is to understand the exact requirements of the customer regarding the product to be developed and to document them properly.

This phase consists of two distinct activities:

Requirements gathering and analysis. Requirements specification.

Requirements Gathering and Analysis

This activity consists of first gathering the requirements and then analyzing the gathered requirements. The goal of the requirements gathering activity is to collect all relevant information regarding the product to be developed from the customer with a view to clearly understand the customer requirements.

Requirements Specification

The customer requirements identified during the requirement gathering and analysis activity are organized into a software requirement specification (SRS) document. The requirements describe the “what” of a system, not the “how”. This document written in a natural language contains a description of

Design

The goal of this phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. Two distinctly different design approaches are being used at present. These are:

- Traditional design approach
- Object-oriented design approach

Traditional Design Approach

The traditional design technique is based on the data flow oriented design approach.
The design phase consists of two activities:

1. first a structured analysis of the requirements specification is carried out,
2. second structured design activity.

Object-Oriented Design Approach

In this technique various objects that occur in the problem domain and the solution domain are identified and the different relationships that exist among these objects are identified.

Coding and Unit Testing

The purpose of the coding and unit testing phase of software development is to translate the software design into source code.

Integration and System Testing

During the integration and system testing phase the different modules are integrated in a planned manner.System testing usually consists of three different kinds of testing activities:

- α –testing: α testing is α testing is the system testing performed by thedevelopment team.
- β –testing: This is the system testing performed by a friendly set of customers.

Maintenance

Software maintenance is a very broad activity that includes error correction, enhancement of capabilities and optimization. Maintenance involvesperforming the following activities:

•**Corrective Maintenance**

This type of maintenance involves correcting error that were not discovered during the product development phase.

•**Perfective Maintenance**

This type of maintenance involves improving the implementation of the system and enhancing the functionalities of the system according to the customer's requirements.

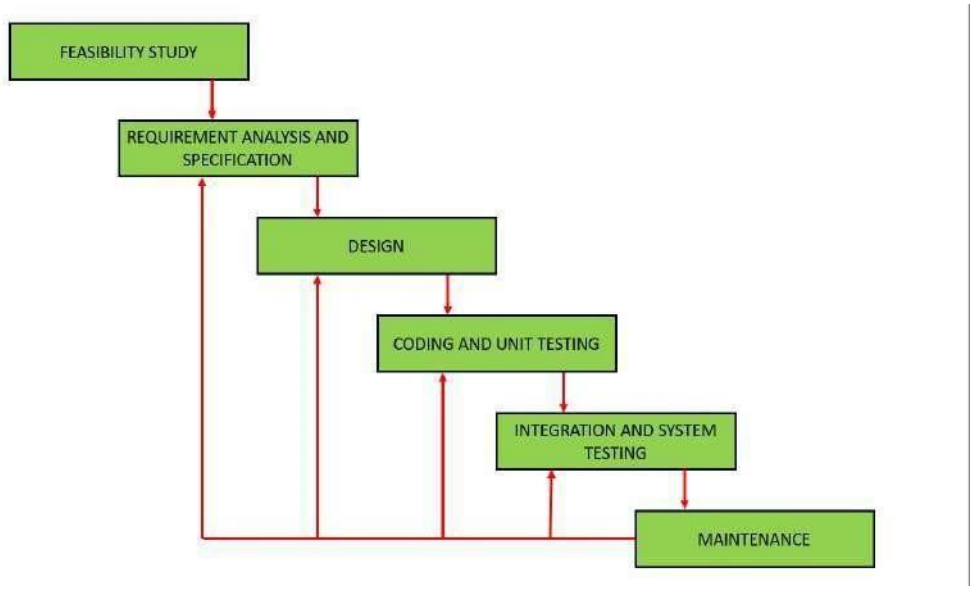
•**Adaptive Maintenance**

Adaptive maintenance is usually required for reporting the softer to work in a new environment.

ITERATIVE WATERFALL MODEL

In a practical software development project, theclassicalwaterfallmodelis hard to use. So, 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.

The iterative waterfall model provides feedback paths from every phase to its preceding phases, which is the main difference from the classical waterfall model.



When errors are detected at some later phase, these feedback paths allow correcting errors committed by programmers during some phase.
Phase Containment of Errors: The principle of detecting errors as close to their points of commitment as possible is known as Phase containment of errors.

Advantages of Iterative Waterfall Model

- **Feedback Path:** In the classical waterfall model, there are no feedback paths, so there is no mechanism for error correction.
- **Simple:** Iterative waterfall model is very simple to understand and use. That'swhy it is one of the most widely used software development models.

EVOLUTIONARY MODEL

Evolutionary model is a combination of Iterativeand Incremental modelof software development life cycle. It is better for software products that have their feature sets redefined during development because of user feedback and other factors. The

Evolutionary development model divides the development cycle into smaller, incremental waterfall models in which users are able to get access to the product at the end of each cycle.

Application of Evolutionary Model:

1. It is used in large projects where you can easily find modules for incremental implementation. Evolutionary model is commonly used when the customer wants to start using the core features instead of waiting for the full software.

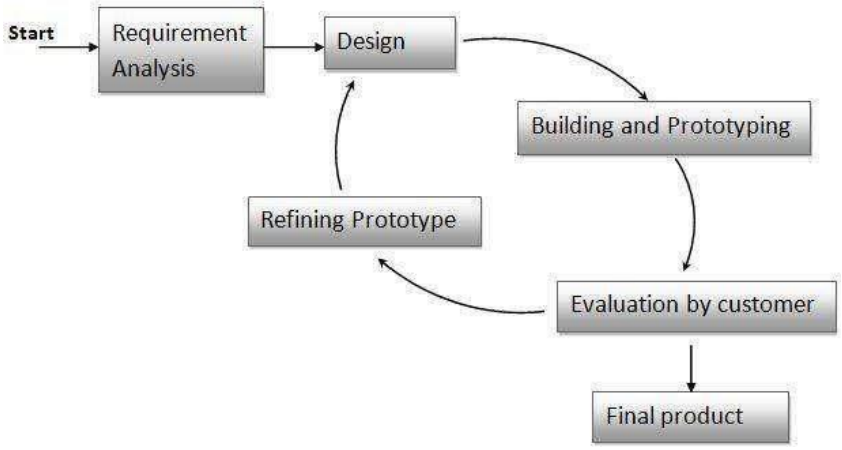
Advantages:

- In evolutionary model, a user gets a chance to experiment partially developed system.
- It reduces the error because the core modules get tested thoroughly.

PROTOTYPING MODEL

Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help to determine the requirements. The main principle of prototyping model is that the project is built quickly to demonstrate the customer who can give more inputs and feedback. This model will be chosen

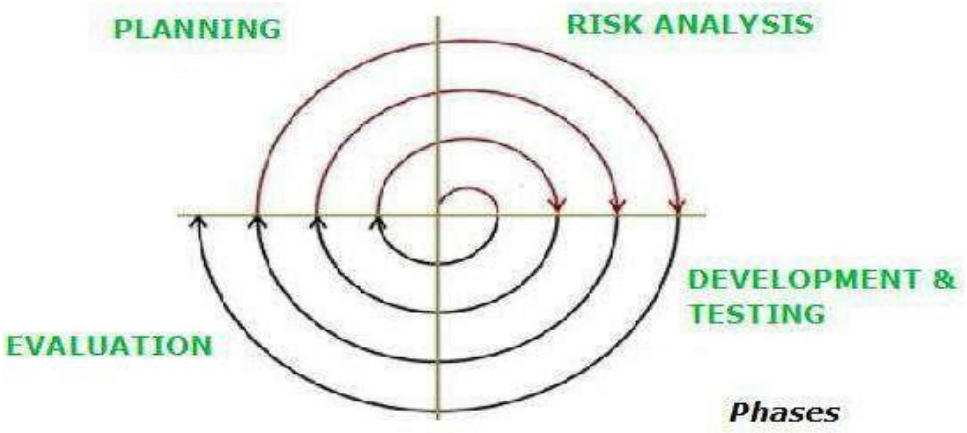
- When the customer defines a set of general objectives for software but does not provide detailed input, processing or output requirements.
- Developer is unsure about the efficiency of an algorithm or the new technology is applied.



Spiral Model

1. Spiral Model is one of the oldest form of the**Software Development LifeCycle Models(SDLC)**, which was first defined by the**Barry Boehm**in the year 1986.

2. Basically, this model is an evolutionary type model, which works on the combined approach of the**waterfall**and**iterativemodel**.



Phases of Spiral Model:

1. Planning:

This phase, mainly consists of following activities:

- Gathering of requirements through consistent interaction with the client and stakeholders.
- Feasibility study.

2. Risk Analysis:-

This is the crucial stage and needs to be carried out attentively, in order to identify all the potential risks, associated with the software product.

Development &Test:-

It is an important phase of this model, where all the requirements, strategies and plans are implemented and executed, so as to develop the software product. Further, the software development process is subsequently followed by **testing activities**.

Evaluation:-

This phase involves the software product interaction with the customers, who assess them and accordingly, provide their feedbacks, which helps in determining the requirements or

features that needs to be added or removed from the software, in the next iteration, so as to satisfy the customer's need.

Spiral Model Strengths

1. Provides early indication of risks, without much cost.
2. Critical high-risk functions are developed first.

Spiral Model Weaknesses

1. The model is complex.
2. Risk assessment expertise is required.

CHAPTER-2
SOFTWARE PROJECT MANAGEMENT

The main goal of software project management is to enable a group of software engineers to work efficiently towards successful completion of the project.

The management of software development is dependent on four factors:

•The People • The Product • The Process • The Project

JOB RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER

- Software managers are responsible for planning and scheduling project development. Manager must decide what objectives are to be achieved, what resources are required to achieve the objectives, how and when the resources are to be acquired and how the goals are to be achieved.

SKILLS NECESSARY FOR SOFTWARE PROJECT MANAGEMENT

- Good qualitative judgment and decision-making capabilities
- Good knowledge of latest software project management techniques such as cost estimation, risk management, configuration management.

PROJECT PLANNING

Software managers are responsible for planning and scheduling project development. They monitor progress to check that the development is on time and within budget. Project planning consists of the following activities:

- Estimate the size of the project.
- Estimate the cost and duration of the project. Cost and duration estimation is usually based on the size of the project. Estimate how much effort would be required?

METRICS FOR PROJECT SIZE ESTIMATION

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

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

1. **Lines of code or LOC**

2. **Function point**

LINES OF CODE

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

Shortcomings of LOC

- LOC is language dependent. A line of assembler is not the same as a line of COBOL.
- LOC measure correlates poorly with the quality and efficiency of the code.

FUNCTION POINT METRICS

- Function point metrics overcomes many of the shortcomings of LOC.Function point metrics proposes that size of the software project is directly dependent on various functionalities it supports. More the features supported the more would be the size.
- Function point metric estimates the size of a software product directly fromthe problem specification.

The different parameters are:

- **Number Of Inputs:**

Each data item input by the user is counted.

- **Number Of Outputs:**

The outputs refers to reports printed, screen outputs, error messages produced etc.

- **Number Of Inquiries:**

It is the number of distinct interactive queries which can be made by the users.

- **Number Of Files:**

Each logical file is counted. A logical file means groups of logically related data. Thus logical files can be data structures or physical files.

- **Number Of Interfaces:**

Here the interfaces which are used to exchange information with other systems

FP = UFP (Unadjusted Function Point) * TCF (Technical Complexity Factor) UFP = (Number of inputs) * 4 + (Number of outputs) * 5 + (Number of inquiries) *4 + (Number of files) * 10 + Number of interfaces) * 10 TCF = DI (Degree of Influence) * 0.01

The unadjusted function point count (UFP) reflects the specific countable functionality provided to the user by the project or application.

Each of these 14 factors is assigned a value from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). Now, the TCF is computed as (0.65+0.01*DI).

As DI can vary from 0 to 70, the TCF can vary from 0.65 to 1.35. Finally
FP = UFP *TCF

Feature Point Metric

Feature point metric incorporates an extra parameter in to algorithm complexity. This parameter ensures that the computed size using the feature point metric reflects the fact that the more the complexity of a function

PROJECT ESTIMATION TECHNIQUES

The estimation of various project parameters is a basic project planning activity. The project parameters that are estimated include:

- Project size(i.e. size estimation)
- Project duration

There are three broad categories of estimation techniques:

- Empirical estimation techniques
- Heuristic techniques
- Analytical estimation techniques

•EMPIRICAL ESTIMATION TECHNIQUES

Empirical estimation techniques are based on making an educated guess of the project parameters. While using this technique, prior experience with the development of similar products is useful.

EXPERT JUDGMENT TECHNIQUE

The most widely used cost estimation technique is the expert judgment, which is an inherently top-down estimation technique. In this approach makes an educated guess of the problem size after analyzing the problem thoroughly. The expert estimates the cost of the different modules or subsystems and then combines them to arrive at the overall estimate.

DELPHI COST ESTIMATION

Delphi cost estimation approach tries to overcome some of the short comings of the expert judgment approach. Delphi estimation is carried out by a team consisting of a group of experts and a coordinator.

A coordinator provides each estimator with the software requirement specification (SRS) document and a form for recording a cost estimate.

- The coordinator prepares and distributes a summary of the estimator's responses and includes any unusual rationales noted by the estimators.

HEURISTIC TECHNIQUES

Heuristic techniques assume that the relationships among the different project parameters can be modelled using suitable mathematical expressions..

Different heuristic estimation models can be divided into two categories:

- Single variable model
- Multivariable model

A single variable estimation model takes the following form:

Estimated parameter = c1* ed1

Where e is a characteristics of the software, c1 and d1 are constants.

A multivariable cost estimation model takes the following form: Estimated Resource = c1 * e1

d1 + c2 * e2 d2

+ .....

Where e1, e2 .. are the basic characteristics of the software. c1, c2,

d1, d2 .. are constants.

COCOMO MODEL

COCOMO was proposed by Boehm. Boehm postulated that any software development project can be classified into one of the following three categories based on the development complexity:

ORGANIC,

SEMIDETACHED,

EMBEDDED.

- **Organic:** In the organic mode the project deals with developing a well- understood application program. The size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.
- **Semidetached:** In the semidetached mode the development team consists of a mixture of experienced and inexperienced staff
- **Embedded:** In the embedded mode of software development, the project has tight constraints, which might be related to the target processor and its interface with the associated hardware. According to Boehm, software cost estimation should be done through three

stages: basic COCOMO, intermediate COCOMO, and complete COCOMO.

Basic COCOMO

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

Effort = a1 × (KLOC)a2 PM
Tdev = b1 × (Effort) b2 Months Where
(i) KLOC is the estimated size of the software product expressed in KiloLines of Code,
(ii) a1, a2, b1, b2 are constants for each category of software products,

Estimation of development effort:

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

Organic: **Effort = 2.4(KLOC)1.05 PM** Semi-Detached: **Effort = 3.0(KLOC)1.12 PM**
Embedded: **Effort = 3.6(KLOC)1.20 PM PM: Person Months**

Estimation of development time:

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

Organic: **Tdev = 2.5(Effort)0.38 Months** Semi-detached: **Tdev = 2.5(Effort)0.35 Months**
Embedded: **Tdev = 2.5(Effort)0.32 Months**

Solution: The basic COCOMO equation takes the form:

Effort=a1*(KLOC) a2 PM
Tdev=b1*(efforts)b2 Months
Estimated Size of project= 400 KLOC

(i)Organic Mode

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

(ii)Semidetached Mode

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

(iii)Embedded Mode

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

Intermediate COCOMO

The basic COCOMO model allowed for a quick and rough estimate, but it resulted in a lack of accuracy. Basic model provides single-variable (software size) static estimation based on the type of the software. A host of the other project parameters besides the product size affect the effort required to develop the product as well as the development time .

The cost drivers are grouped into four categories:

- Product attributes
- Computer attributes
- Personnel attributes
- Development environment

Product

The characteristics of the product data considered include the inherent complexity of the product, reliability requirements of the product, database size etc.

Computer

The characteristics of the computer that are considered include the execution speed required, storage space required etc.

Personnel

The attributes of development personnel that are considered include the experience level of personnel, programming capability, analysis capability etc.

Development Environment

The development environment attributes capture the development facilities available to the developers.

Complete COCOMO / Detailed COCOMO

Basic and intermediate COCOMO model considers a software product as a single homogeneous entity. Most large system are made up of several smaller subsystem. These subsystems may have widely different characteristics.

Some subsystem may be considered organic type, some embedded and someemidetached. Software development is executed in different phases and hence the estimation of efforts and schedule of deliveries should be carried out phase wise.

ANALYTICAL ESTIMATION TECHNIQUES

Analytical estimation techniques derive the required results starting with certain basic assumptions regarding the project. This technique does have a scientific basis.

Halstead's Software Science an Analytical Estimation Techniques

Halstead's software science is an analytical technique to measure size, development effort, and development cost of software products. Halstead used a few primitive program parameters to develop the expressions for the overall program length, potential minimum volume, language level,

Operators and Operands for the ANSI C Language

The following is a suggested list of operators for the ANSI C language:

({ . , -> * + - ~ ! ++ -- * / % + - <<>><><= >= != == & ^ | && \\ = *=
/= %= -= <<= >>= &= ^= \= : ? { ; CASE DEFAULT IF ELSE
SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN
and a function name in a function call.

Length and Vocabulary

The length of a program as defined by Halstead, quantifies the total usage of all operations and operands in the program. Thus, length N = N1 + N2

The program vocabulary is the number of unique operators and operands used in the program. Thus, program vocabulary $\eta = \eta 1 + \eta 2$

Program Volume

The length of a program depends on the choice of the operators and operands used.

$V = N \log_2 \eta$

The program volume V is the minimum number of bits needed to encode the program. In fact, to represent η different identifiers uniquely, we need at least $\log_2 \eta$ bits. We need N $\log_2 \eta$ bits to store a program of length N.

Effort and Time

The effort required to develop a program can be obtained by dividing the program volume by the level of the programming language used to develop the code. Thus, effort E = V / L, where E is the number of mental discriminations required to implement the program and also the effort required to read and understand the program.

Actual Length Estimation

Even though the length of a program can be found by calculation the total number of operators and operands in a program.

$$N = \eta 1 \log_2 \eta 1 + \eta 2 \log_2 \eta 2$$

Project Scheduling

Project-task scheduling is a significant project planning activity. It comprises deciding which functions would be taken up.

Identify all the functions required to complete the project.

1. Break down large functions into small activities.
2. Determine the dependency among various activities.
3. Establish the most likely size for the time duration required to complete the activities.

Work Breakdown Structure

Most project control techniques are based on breaking down the goal of the project into several intermediate goals. Each intermediate goal can be broken down further. This process can be repeated until each goal is small enough to be well understood. Work breakdown structure (WBS) is used to decompose a given task set recursively into small activities. In this technique, one builds a tree whose root is labeled by the problem name.

Activity Networks and Critical Path Method

Work Breakdown Structure representation of a project is transformed into an activity network by representing the activities identified in work breakdown structure along with their interdependencies. An activity network shows the different activities making up a project, their estimated durations and interdependencies.

Critical Path Method

From the activity network representation, the followinganalysis can be made:

- The minimum time (MT) to complete the project is the maximum of all paths from start to finish.
- The earliest start (ES) time of a task is the maximum of all paths from the start to this task.
- The latest start (LS) time is the difference between MT and the maximum of all paths from this task to the finish.

A path from the node to the finish node containing only critical tasks is called a critical path.

GANTT CHART

A Gantt chart, commonly used in project management, is one of the most popular and useful ways of showing activities (tasks or events) displayed against time. On the left of the chart is a list of the activities and along the top is a suitable time scaleWhat the various activities are

- When each activity begins and ends
- How long each activity is scheduled to last

Organization structure:

Usually, each software package development organization handles many projects at any time. oftware package organizations assign totally different groups of engineers to handle different software projects.

There are basically 2 broad ways in which a software package development organization is structured: *Project format, and Functional format*. These are explained as following below.

1. **Project format:**

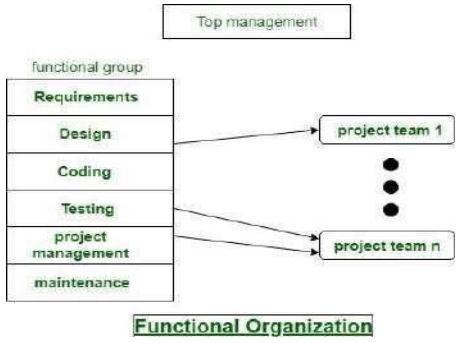
The project development workers are divided supported the project that they work . In the project format, a group of engineers is appointed to the project at the beginning of the

2. **Functional format:**

In the functional format, totally different groups of programmers perform different phases of a project. For example, one team may do the necessities specification, another do the planning, and so on.

Team Structure

project. This needs sensible quality documentation to be made when each activity.



Functional Organization

Chief Programmer Team In this organization, a senior engineer provides the technical leadership and is designated as the chief programmer. The chief programmer partitions the task into small activities and assigns them to the team members.

The chief programmer team is probably the most efficient way of completing and small projects

Democratic Team

The democratic team structure does not enforce any formal team hierarchy.

Typically a manager provides the administrative leadership. At different

Mixed Control Team Organization

The mixed team organization draws upon the ideas from both the democratic organization and the chief programmer organization. This team organization incorporates both hierarchical reporting and democratic set-up. The mixed control team organization is suitable for large team sizes

CHARACTERISTICS OF A GOOD SOFTWARE ENGINEER

The attributes that smart package engineers ought to posses are as follows:

- Exposure to systematic techniques, i.e., familiarity with package engineering principles.
- Smart technical data of the project areas (Domain knowledge).
- Smart programming talents.
- Smart communication skills. These skills comprise of oral, written, and interpersonal skills.

Importance of Risk Identification, Risk Assessment and Risk containment with reference to Risk Management

Risk management is an emerging area that aims to address the problem of identifying and managing the risk associated with a software project. It is really good to identify it,

Risk management consist of three essential activities:

·Risk identification

·Risk assessment

·Risk containment

Risk Identification

A project can get affected by a large variety of risks. Risk identification identifies all the different risks for a particular project. In order to identify the important risks which might affect a project.

- **Project Risks**

Project risks concern various forms of budgetary, schedule, personal, resource and customer- related problems. Software is intangible, it is very difficult to monitor and control a software project.

- **Technical Risks**

Technical risk concern potential design, implementation, interfacing, testing, and maintenance problem. Technical risks also include incomplete specification, changing specification, technical uncertainly.

•**Business risks**

Business risks include risks of building an excellent product that no one wants, losing budgetary or personal commitments etc.

Risks Assessment

The goal of risks assessment is to rank the risks so that risk management can focus attention and resources on the more risks items. For risks assessment,each risk should be rated in two ways:

Risk Containment

After all the identified risk of a project is assessed, plans must be made to contain the most damaging and the most likely risks. Three main strategies used for risks containment are:

- Avoid the risk
- Risk reduction
- Transfer the risk

Avoid the Risk

This may take several forms such as discussions with the customer to reduce the scope of the work and giving incentives to engineers to avoid the risk of manpower turnover etc.

Transfer the Risk

This strategy involves getting the risky component develops by a third party or buying insurance cover etc.

Risk Reduction

This involves planning ways to contain the damage due to a risk.

Risk leverage = (risk exposure before reduction – risk exposure after reduction) / (Cost of reduction)

SOFTWARE CONFIGURATION MANAGEMEnt

Configuration Management helps organizations to systematically manage, organize, and control the changes in the documents, codes, and other entities during the Software Development Life Cycle. It is abbreviated as the SCM process

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

•There are multiple people working on software which is continually updating

- It may be a case where multiple version, branches, authors are involved in a software project, and the team is geographically distributed and works concurrently
- Changes in user requirement, policy, budget, schedule need to be accommodated.

Baseline:

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

Activities during this process:

- Facilitate construction of various versions of an application
- Defining and determining mechanisms for managing various versions of these work products

Change Control:

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

Activities during this process:

- Control ad-hoc change to build stable software development environment. Changes are committed to the repository
- The request will be checked based on the technical merit, possible side effects and **Configuration Status Accounting:**

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

Activities during this process:

- Keeps a record of all the changes made to the previous baseline to reach a new baseline
- Identify all items to define the software configuration

CHAPTER-3
REQUIREMENTS ANALYSIS AND SPECIFICATION

The requirements analysis and specification phase starts once the feasibility study phase is completed and the project is found to be financially sound and technically feasible. The goal of the requirement analysis and specification phase is to clearly understand the customer requirements and to systematically organize these requirements in a specification document. This phase consists of two activities:

- Requirements gathering and analysis.
- Requirements specification

REQUIREMENTS GATHERING AND ANALYSIS

The analyst starts requirements gathering and analysis activity by collecting all information from the customer which could be used to develop the requirements of the system.

Two main activities involved in the requirements gathering and analysis phase are:

- Requirements Gathering: The activity involves interviewing the end users and customers and studying the existing documents to collect all possible information regarding the System.

  data output by the system?

- What are the likely complexities that might arise while solving the problem?

After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various requirements problems

anomalies,

- inconsistencies, •incompleteness.

Anomaly: An anomaly is an ambiguity in the requirement. When a requirement is anomalous, several interpretations of the requirement are possible.

Inconsistency: Two requirements are said to be inconsistent, if one of the requirements contradicts the other two-end user of the system give inconsistent description of the requirement

Incompleteness: An incomplete set of requirements is one in which some requirements have been overlooked

SOFTWARE REQUIREMENT SPECIFICATION

After the analyst has collected all the required information regarding the software to be developed and has removed all incompleteness, inconsistencies and anomalies from the specification, analyst starts to systematically organize the requirements in the form of an SRS document.

Some of the important categories of users of the SRS document and their needs are as follows.

- **Users, customers and marketing personnel**

  The goal of this set of audience is to ensure that the system as describe in the SRS document

- **The software developers refer to the will meet their needs.**

  SRS document to make sure that they develop exactly what is required by the customer.

- **Test Engineers**: Their goal is to ensure that the requirements are

understandable from a functionality point of view, so that they can test the software and validate its working.

- **User Documentation Writers**: Their goal in reading the SRS document is to ensure that they understand the document well enough to be able to write the users' manuals.

Project Managers

They want to ensure that they can estimate the cost of the project easily by referring to be SRS document and that it contains all information required to plan the project.

Maintenance Engineers

The SRS document helps the maintenance engineers to understand the functionalities of the system.

CONTENTS OF THE SRS DOCUMENT

An SRS document should clearly document:

Functional Requirements Non functional Requirements Goals of implementation

Functional Requirement

The functional requirements of the system as documented in the SRS document should clearly describe each function which the system would support along with the corresponding input and output data set.

NONFUNCTIONAL REQUIREMENTS:-

Non-functional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.

GOALS OF IMPLEMENTATION:-

The goals of implementation part documents some general suggestionsregarding development. These suggestions guide trade-off among design goals.The goals of implementation section might document issues such as revisions tothe system functionalities that may be required in the future, new devices to besupported in the future, reusability issues, etc.

DOCUMENTING FUNCTIONAL REQUIREMENTS

For documenting the functional requirements, we need to specify the set offunctionalities supported by the system. A function can be specified by identifyingthe state at which the data is to be input to the system, its input data domain, theoutput data domain, and the type of processing to be carried on the input data toobtain the output data

PROPERTIES OF A GOOD SRS DOCUMENT

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

CONCISE. The SRS document should be concise and at the same timeunambiguous, consistent, and complete. Verbose and irrelevantdescriptions reduce readability and also increase error possibilities.

STRUCTURED. It should be well-structured. A well-structured document iseasy to understand and modify. In practice, the SRS documentundergoes several revisions to cope up with the customerrequirements. Often, the customer requirements evolve over a periodof time. Therefore, in order to make the modifications to the SRSdocument easy, it is important to make the document well-structured.

BLACK-BOX VIEW. It should only specify what the system should do andrefrain from stating how to do these. This means that the SRSdocument should specify the external behaviour of the system and notdiscuss the implementation issues. For this reason, the SRSdocument is also called the black-box specification of a system.

CONCEPTUAL INTEGRITY. It should show conceptual integrity so that thereader can easily understand it.

RESPONSE TO UNDESIRED EVENTS. It should characterize acceptableresponses to undesired events. These are called system response toexceptional conditions.

VERIFIABLE. All requirements of the system as documented in the SRSdocument should be verifiable. This means that it should be possible todetermine whether or not requirements have been met in animplementation.

ORGANIZATION OF THE SRS DOCUMENT

Organization of the SRS document and the issues depends on the type of the product being developed. Three basic issues of SRS documents are: functional requirements, non functional requirements, and guidelines for system implementations. The SRS document should be organized into:

1. Introduction

(a) Background

(b)Overall Description

(c)Environmental Characteristics

Goals of implementation
Functional requirements
Non-functional Requirements
Behavioural Description
(a)System States
(b)Events and Actions

The **`introduction'** section describes the context in which the system is being developed, identify the purpose of your product.

an overall description of the system an overview of the product you build before listing specifications.

The environmental characteristics.

The environmental characteristics subsection describes the properties of the environment with which the system will interact.

Goals of implementation

The goals of implementation section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues,

Functional requirements-Functional requirements outline the system's behaviour or WHAT it should do under different circumstances and in various use scenarios.

Nonfunctional Requirements

- Reliability Maintainability Portability

Behavioral Description

Specification of behaviour may or may not be necessary for all system.it is usually necessary for those system in which the system behaviour depends on the state in which the system is and the system transit among a set of states depending on some prespecified condition and event.

TECHNIQUES FOR REPRESENTING COMPLEX LOGIC:-

Good SRS documents sometimes may have the conditions which are complex &

which may have overlapping interactions & processing sequences. There are two main techniques available to analyze& represent complex processes logic are

A)Decision tree

B)Decision table

DECISION TREE

A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken. The edges of a decision tree represent conditions and the leaf nodes represent the actions to be performed depending on the outcome of testing the condition.

Example: - Consider Library Membership Automation Software (LMS) where it should support the following three options:

• **New member • Renewal • Cancel membership New member option**

Decision: When the 'new member' option is selected, the software asks details about the member like the member's name, address, phone number etc.

Action: If proper information is entered then a membership record for the member is created and a bill is printed for the annual membership charge plus the security deposit payable.

Renewal option

Decision: If the 'renewal' option is chosen, the LMS asks for the member's name and his membership number to check whether he is a valid member or not.

Cancel membership option

Decision: If the 'cancel membership' option is selected, then the software asks for member's name and his membership number.

CHAPTER-4
SOFTWARE DESIGN

Software design and its activities Software design deals with transforming the customer requirements, as described in the SRS document, into a form (a set of documents) that is suitable for implementation in a programming language.Design activities can be broadly classified into two important parts:

• Preliminary (or high-level) design and

• Detailed design.

Preliminary and detailed design activities

The meaning and scope of two design activities (i.e. high-level and detailed design) tend to vary considerably from one methodology to another.

High-level design means identification of different modules and the control relationships among them and the definition of the interfaces among these modules. The outcome of high-level design is called the program structure or software architecture. Many different types of notations have been used to represent a high- level design.

During **detailed design**, the data structure and the algorithms of the different modules are designed. The outcome of the detailed design stage is usually known as the module-specification document.

Items developed during the software design phase For a design to be easily implemented in a conventional programming language

CHARACTERISTICS OF A GOOD SOFTWARE DESIGN

The definition of "a good software design" can vary depending on the application being designed. For example, the memory size used by a program may be an important issue to characterize a good solution for embedded software development – since embedded applications are often required to be implemented using memory of limited size due to cost, space, or power consumption considerations. For embedded applications, one may sacrifice design comprehensibility to achieve code compactness.

The characteristics are listed below:

- **Correctness:** A good design should correctly implement all the functionalities identified in the SRS document.
- **Understandability**: A good design is easily understandable.
- **Efficiency:** It should be efficient. •**Maintainability:** It should be easily amenable to change.

Features of a design document In order to facilitate understandability, the design should have the following features: • It should use consistent and meaningful names for various design components.

- The design should be modular. The term modularity means that it should use a cleanly decomposed set of modules.

Modularity

A modular design achieves effective decomposition of a problem. It is a basic characteristic of any good design solution. Decomposition of a problem into modules facilitates the design by taking advantage of the divide and conquers principle

Clean Decomposition

The modules in a software design should display high cohesion and low coupling. The modules are more or less independent of each other.

Layered Design

In a layered design, the modules are arranged in a hierarchy of layers. A module can only invoke functions of the modules in the layer immediately below it. A layer design can make the design solution easily understandable.

COHESION

Most researchers and engineers agree that a good software design implies clean decomposition of the problem into modules, and the neat arrangement of these modules in a hierarchy. The primary characteristics of neat module decomposition are high cohesion and low coupling. Cohesion is a measure of functional strength of a module

Error isolation Functional independence reduces error propagation. If a module is functionally independent, its degree of interaction with other modules is less. Therefore, any error existing in a module would not directly affect the other modules.

Scope for Reuse- Reuse of a module becomes possible, because each module does some welldefined and precise functions and the interface of the module with other module is simple and minimal.

Understandability Complexity of the design is reduced, because different modules are more or less independence of each other and can be understood in isolation.

Low High **CLASSIFICATION OF COHESION**

COINCIDENTAL COHESION: A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, the module contains a random collection of functions. It is likely that the functions have been put in the module out of pure coincidence without any thought or design.

LOGICAL COHESION:A module is said to be logically cohesive, if all elements of the module perform similar operations, e.g. error handling, data input, data output, etc

TEMPORAL COHESION: When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion

PROCEDURAL COHESION:A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective.

COMMUNICATIONAL COHESION: A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure,

,

FUNCTIONAL COHESION: Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function. For example, a module containing all the functions required to manage employees' pay-roll exhibits functional

CLASSIFICATION OF COUPLING

Data coupling: Two modules are data coupled, if they communicate through a parameter. An example is an elementary data item passed as a parameter between two modules.

Stamp coupling: Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C. Control coupling: **Control coupling**exists between two modules, if data from one module is used to direct the order of instructions execution in another

Common coupling: Two modules are common coupled, if they share data through some global data items.

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

S/W Design Approaches Two different approaches to software design are: Function-oriented design and

Object-oriented design

Function oriented design Features of the function-oriented design approach are: Top-down decomposition In top-down decomposition, starting at a high-level view of the system, each high-level function is successfully refined into more detailed functions.

This function may consists of the following subfunctions: •assign-membership-number

• create-member-record

• print-bill Each of these sub functions may be split into more detailed sub functions and so on.

Object Oriented Design :

In the object-oriented design approach, the system is viewed as a collection of objects. The system state is decentralized among the objects and each object manages its own state information.

STRUCTURED ANALYSIS METHODOLOGY

The aim of structured analysis activity is to transform a textual problem description into a graphic model. Structured analysis is used to carry out the top-down decomposition of the set of high-level functions depicted in the problem description and to represent them graphically. Top-down decomposition approach

- Divide and conquer principle. Each function is decomposed independently
- Graphical representation of the analysis results using Data Flow Diagram (DFD).

USE OF DATA FLOW DIAGRAM

The DFD also known as bubble chart is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on these data & the output data generated by the system. DFD is a very simple formalism – it is simple to understand and use.

LISTS THE SYMBOLS USED IN DFD

Five different types of primitive symbols used for constructing DFDs. The

meaning of each symbol is

Functional symbol ( ) : A function is represented is using a circle.
External entity symbo) : An external entities are essentially those physl (ical entities external to the software system which interact with the system by inputting data to the system or by consuming the data produced by the system.
Data flow symbol () : A directed arc or an arrow is used as a data flow symbol.
Data store symbol ( ) : A data store represents a logical file. It is represented using two parallel lines.

Output symbol ( ) : The output symbol is used when a hard copy is produced and the user of the copies cannot be clearly specified or there are several users of the output.

CONSTRUCTION OF DFD

A DFD model of a system graphically represent how each input data is transformed to its corresponding output data through a hierarchy of DFDs. A DFD start with the most abstract definition of the system (lowest level) and at each higher level DFD

CONTEXT DIAGRAM

The context diagram represents the entire system as a single bubble. The bubble is labelled according to the main function of the system. The various external entities with which the system interacts and the data flows occurring between the system and the external entities are also represented.

LEVEL 1 DFD

The level 1 DFD usually contains between 3 and 7 bubbles. To develop the Level 1 DFD, examine the high-level functional requirements. If there are between 3 to 7 high-level functional requirements, then these can be directly represented as bubbles in the Level 1 DFD

DECOMPOSITION

STRUCTURED DESIGN

The aim of structured design is to transform the results of the structured analysis (i.e. a DFD representation) into a structure chart. Structured design provides two strategies to guide transformation of a DFD into a structure chart.

- Transform analysis
- Transaction analysis

STRUCTURE CHART :

A structure chart represents the software architecture, i.e. the various modules making up the system, the dependency (which module calls which other modules), and the parameters that are passed among the different modules. The basic building blocks which are used to design structure charts are the following:

- Rectangular boxes: Represents a module.
- Module invocation arrows: Control is passed from one module to another module in the direction of the connecting arrow.
- Data flow arrows: Arrows are annotated with data name; named data passes from one module to another module in the direction of the arrow.

TRANSFORM ANALYSIS

Transform analysis identifies the primary functional components (modules) and the high level inputs and outputs for these components. The first step in transform analysis is to divide the DFD into 3 types of parts:

- Input
- Logical processing
- Output

The input portion in the DFD includes processes that transform input data fr om physical to logical form. Each input portion is called an afferent branch. The output portion of a DFD transforms output data from logical form to physical form. Each output portion is called an efferent branch.

Transaction Analysis

A transaction allows the user to perform some meaningful piece of work. In a transaction-driven system, one of several possible paths through the DFD is traversed depending upon the input data item. Each different way in which input data is handled in a transaction. The number of bubbles on which the input data to the DFD are incident defines the number of transactions.

CHAPTER-5

USER INTERFACE DESIGN

Characteristics of a user interface It is very important to identify the characteristics desired of a good user interface. Because unless we are aware of these, it is very much difficult to design a good user interface. A few important characteristics of a good user interface are the following:

•Speed of learning. A good user interface should be easy to learn. Speed of learning is hampered by complex syntax and semantics of the command issue procedures. A good user interface should not require its users to memorize commands. Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface. Besides, the following three issues are crucial to enhance the speed of learning:

f Use of Metaphors and intuitive command names.

Speed of learning an interface is greatly facilitated if these are based on some day-to-day real-life examples or some physical objects with which the users are familiar. The abstractions of real-life objects or concepts used in user interface design are called metaphors. If the user interface of a text editor uses concepts similar to the tools used by a writer for text editing such as cutting lines and paragraphs and pasting it at other places, users can immediately relate to it. Another popular metaphor is a shopping cart.

. *f* Consistency. Once a user learns about a command, he should be able to use the similar commands in different circumstances for carrying out similar actions. This makes it easier to learn the interfacesince the user can extend his knowledge about one part of the interface to the other parts

f Component-based interface. Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications with which the user is already familiar. This can be achieved if the interfaces of different applications are

developed using some standard user interface components. This, in fact, is the theme of the component-based user interface

• Attractiveness. A good user interface should be attractive to use. An attractive user interface catches user attention and fancy. In this respect, graphics- based user interfaces have a definite advantage over text-based interfaces.

• Consistency. The commands supported by a user interface should be consistent. The basic purpose of consistency is to allow users to generalize the knowledge about aspects of the interface from one part to another.

• Feedback. A good user interface must provide feedback to various user actions. Especially, if any user request takes more than few seconds to process, the user should be informed about the state of the processing of his request. In the absence of any response from the computer for a long time, a novice user might even start recovery/shutdown procedures in panic.

• Support for multiple skill levels. A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users. This is necessary because users with different levels of experience in using an application prefer different types of user interfaces. Experienced users are more concerned about the efficiency of the command issue procedureAfter using an application for extended periods of time,

• Speed of recall. Once users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximized. This characteristic is very important for intermittent users. Speed of recall is improved if the interface is based on some metaphors, symbolic command issue procedures, and intuitive command names.

• Error prevention. A good user interface should minimize the scope of committing errors while initiating different commands. The error rate of an interface can be easily determined by monitoring the errors committed by average users while using the interface. This monitoring can be automated by instrumenting the user interface code with monitoring code he becomes familiar with the operation of the software

• Error recovery (undo facility). While issuing commands, even the expert users can commit errors. Therefore, a good user interface should allow a user to undo a mistake committed by him while using the interface. Users are put to inconvenience, if they cannot recover from the errors they commit while using the software.

• Mode-based interface vs. modeless interface - A mode is a 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

• A GUI usually supports command selection using an attractive and user- friendly menu selection system. *ƒ* In a GUI, a pointing device such as a mouse or a light pen can be used for issuing commands. The use of a pointing device increases the efficacy issue procedure.

TYPES OF USER INTERFACE

User interfaces broadly classified into three categories:

Command language-based interfaces

Menu-based interfaces Direct

Direct manipulation interfaces

COMMAND LANGUAGE-BASED INTERFACES

A command language-based interface is based on designing a command language which the user can use to issue the commands. The user is expected to frame the appropriate commands in the language and type whenever required. Command language-based interface allow fast interaction with the computer and simplify the input of complex commands. Obviously, for inexperienced users, command language-based interfaces are not suitable. A command language-based interface is easier to develop compared to a menu-based or a direct-manipulation interface because complier writing techniques are well developed

ISSUES IN DESIGNING A COMMAND LANGUAGE INTERFACE

The designer has to decide what mnemonics to use for the different commands. The designer should try to develop meaningful mnemonics and yet be concise to minimize the amount of typing required.

The designer has to decide whether the user will be allowed to redefine the command names to suit their own preferences.

MENU-BASED INTERFACES

The advantage of a menu-based interface over a command language-based interface is that menu-based interface does not require the users to remember the exact syntax of the commands. A menu based interface is based on recognition of the command names. In this type of interface the typing effort is minimal as most interactions are carried out through menu selections using a pointing device.

SCROLLING MENU When a full choice list cannot be displayed within the menu area, scrolling of the menu items is required. This enables the user to view and select the menu items that cannot be accommodated on the screen.

{DIRECT MANIPULATION INTERFACES: Direct manipulation interfaces present the interface to the user in the form of visual models i.e. icons. This type of interface is called as iconic interface. In this type of interface, the user issues commands by performing actions on the visual representations of the objects.

Main aspects of Graphical UI, Text based Interface Aspects of GUI In a GUI, multiple windows with different information can simultaneously be displayed on the user screen. Iconic information representation and symbolic information manipulation is possible in a GUI. Symbolic information manipulation, such as dragging an icon representing a file to a trash can for deleting, is intuitively very appealing and the user can instantly remember it.

WINDOW

A window is a rectangular area on the screen. A window can be considered to be a virtual screen, in the sense that it provides an interface to the user for carrying out independent activities, e.g. one window can be used for editing a program and another for drawing pictures, etc. A window can be divided into two parts: client part, and non-client part.

WINDOW MANAGER AND WINDOW SYSTEM

Window manager is the component of WMS with which the end user interacts to do various window-related operations such as window repositioning, window resizing, iconification, etc. The window manager is built on the top of the window system in the sense that it makes use of various services provided by the window system. The window manager and not the window system determines how the windows look and behave. In fact, several kinds of window managers can be developed based on the same window system

WINDOW MANAGER

The window manager is responsible for managing and maintaining the non-client area of a window. Window manager manages the real-estate policy, provides look and feel of each individual window.

TYPES OF WIDGETS

(window objects) Different interface programming packages support different widget sets. However, a surprising number of them contain similar kinds of widgets, so that one can think of a generic widget set which is applicable to most interfaces. The following widgets are representatives of this generic class.

LABEL WIDGET

This is probably one of the simplest widgets. A label widget does nothing except to display a label, i.e. it does not have any other interaction capabilities and is not sensitive to mouse clicks. A label widget is often used as a part of other widgets.

CONTAINER WIDGET

These widgets do not stand by themselves, but exist merely to contain other widgets. Other widgets are created as children of the container widget. When the container widget is moved or resized, its children widget also get moved or resized.

PULL-DOWN MENU

These are more permanent and general. You have to move the cursor to a specific location and pull down this type of menu.

Dialog boxes. We often need to select multiple elements from a selection list. A dialog box remains visible until explicitly dismissed by the user. A dialog box can include areas for entering text as well as values

PUSH BUTTON

A push button contains key words or pictures that describe the action that is triggered when you activate the button. Usually, action related to a push button occurs immediately when you click a push button unless it contains an ellipsis (…). A push button with an ellipsis generally indicates that another dialog box will appear.

RADIO BUTTONS

A set of radio buttons is used when only one option has to be selected out of many options. A radio button is a hollow circle followed by text describing the option it stands for. When a radio button is selected, it appears filled and the previously selected radio

button from the group is unselected. Only one radio button from a group can be selected at any time.

COMBO BOXES

A combo box looks like a button until the user interacts with it. When the user presses or clicks it, the combo box displays a menu of items to choose from. Normally a combo box is used to display either one-of-many choices when space is limited, the number of choices is large, or when the menu items are computed at run-time.

X-WINDOW

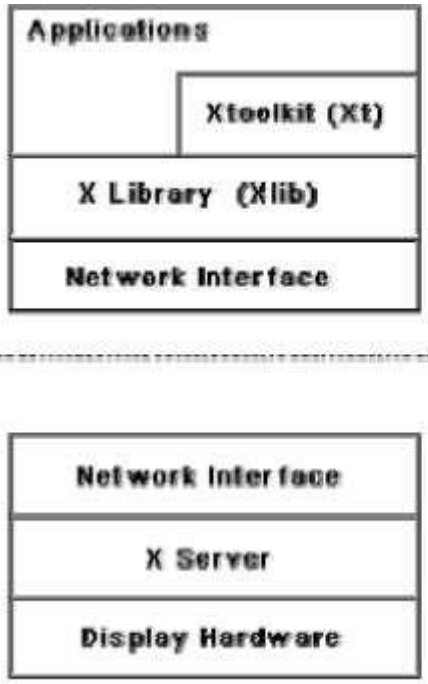
The X-window functions are low-level functions written in C language which can be called from application programs. But only the very serious application designer would program directly using the X-windows library routines. Built on top of X- windows are higher-level functions collectively called Xtoolkit. Xtoolkit consists of a set of basic widgets and a set of routines to manipulate these widgets.

POPULARITY OF X-WINDOW

One of the important reasons behind the extreme popularity of the X-window system is probably due to the fact that it allows development of portable GUIs. Applications developed using X-window system are device-independent. Also, applications developed using the X-window system become network independent in the sense that the interface would work just as well on a terminal connected anywhere on the same network as the computer running the application is. Network-independent

ARCHITECTURE OF AN X-SYSTEM

The X-architecture is pictorially depicted in fig. 9.9. The different terms used in this diagram are explained below.



; and the widgets. We have already seen
onents such as scroll bars, push buttons,
)out a dozen library routines that allow a
user interface. In order to develop a user
t of components (widgets) he needs, and
resources) and behavior of these widgets

by using the intrinsic routines to complete the development of the interface. Therefore, developing an interface using Xtoolkit is much easier than developing the same interface using only X library.

VISUAL PROGRAMMING

Visual programming is the drag and drop style of program development. In this style

CHAPTER-6

CODING AND TESTING

CODING

Good software development organizations normally require their programmers to adhere to some well-defined and standard style of coding called coding standards. Most software development organizations formulate their own coding standards that suit them most, and require their engineers to follow these standards rigorously. A coding standard gives a uniform appearance to the codes written by different engineers.

- It enhances code understanding.
- It encourages good programming practices.

CODING STANDARDS AND GUIDELINES

Good software development organizations usually develop their own coding standards and guidelines depending on what best suits their organization and the type of products they develop.

The following are some representative coding standards.

Rules for limiting the use of global: These rules list what types of data can be declared global and what cannot.

Contents of the headers preceding codes for different modules: The information contained in the headers of different modules should be standard for an organization. The exact format in which the header information is organized in the header can also be specified. The following are some standard header data:

- Name of the module.
- Date on which the module was created.
- Author's name
- Modification history.
- Synopsis of the module.

Naming conventions for global variables, local variables, and constant identifiers: A possible naming convention can be that global variable names always start with a capital letter, local variable names are made of small letters, and constant names are always capital letters.

Error return conventions and exception handling mechanisms: The way error conditions are reported by different functions in a program are handled should be standard within an organization. For example, different functions while encountering an error condition should either return a 0 or 1 consistently.

Do not use a coding style that is too clever or too difficult to understand: Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code. Clever coding can obscure meaning of the code and hamper understanding.

Avoid obscure side effects: The side effects of a function call include modification of parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code.

Do not use an identifier for multiple purposes: Programmers often use the same identifier to denote several temporary entities. For example, some programmers use a temporary loop variable for computing and a storing the final result. The rationale that is usually given by these programmers for such multiple uses of variables is memory efficiencyEach variable should be given a descriptive name indicating its purpose.

The code should be well-documented: As a rule of thumb, there must be at least one comment line on the average for every three-source line. **The length of any function should not exceed 10 source lines**: A function that is very lengthy is usually very difficult to understand as it probably carries out many different functions. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.

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

Aim of testing

The aim of the testing process is to identify all defects existing in a software product. However for most practical systems, even after satisfactorily carrying out the testing phase, it is not possible to guarantee that the software is error free. This is because of the

fact that the input data domain of most software products is very large. It is not practical to test the software exhaustively with respect to each value that the input data may assume

Differentiate between verification and validation.

Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase, whereas validation is the process of determining whether a fully developed system conforms to its requirements specification.

CodeWalk-Through

The main objective of code walk-through is to discover the algorithmic and logical errors in the code. Code walkthrough is an informal code analysis technique.

In this technique, after a module has been coded, it is successfully compiled and all syntax errors are eliminated. Some members of the development team are given the code a few days before the walk- through meeting to read and understand the code.

Some guidelines are:

- The team performing the code walkthrough should not be either too big or too small. Ideally, it should consist of three to sevenmembers.
- Discussions should focus on discovery of errors and not on how to fix the discoverederrors.

The principal aim of code inspection is to check for the presence of some common types of errors caused due to oversight and improper programming. Some classical programming errors which can be checked during code inspection are:

- Use of uninitializedvariables
- Jumps intoloops
- Non-terminatingloops

Software Documentation`

Different kinds of documents such as user's manual, software requirements specification (SRS) document, design document, test document, installation manual are part of the software engineering process. Good documents are very useful and serve the following purposes:

➢ Good documents enhance understandability and maintainability of a software product. They reduce the effort and time required for maintenance.

➢ Good documents help the users in effectively exploiting thesystem.

Different types of software documents can be broadly classified into: ○Internaldocumentation○Externaldocumentation

Internal Documentation

Internal documentation is the code comprehension features provided in the source code itself. Internal documentation can be provided in the code in several forms. The important types of internal documentation are:

- Comments embedded in the sourcecode
- Use of meaningful variablenames
- Module and functionheaders
-

External documentation

External documentation is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test document etc

Distinguishamong Unit Testing, Integration Testing, and SystemTesting

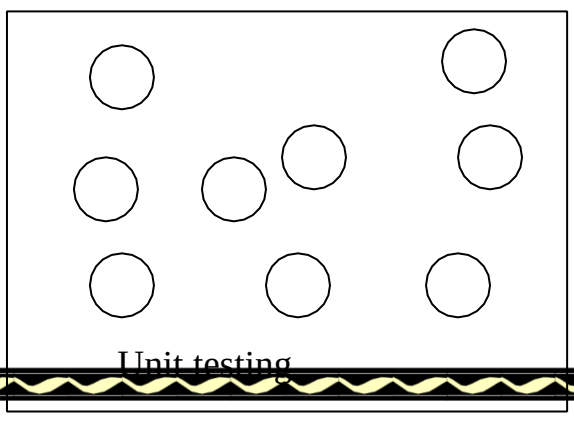
A software product is normally tested in the three levels:

- Unittesting
- Integrationtesting
- Systemtesting

A unit test is a test written by the programmer to verify that a relatively small piece of code is doing what it is intended to do. They are narrow in scope, they should be easy to write and execute, and their effectiveness depends on what the programmer considers to be useful. The tests are intended for the use of the programmer. Unit tests shouldn't have dependencies on outside systems.

UnitTesting

Unit testing or module testing of



Unit testing

different units or modules of a system in isolation.

Fig:6.1 Unit testing

Unit testing is undertaken when a module has been coded and successfully reviewed. The purpose of testing is to find and remove the errors in the software as practical. The numbers of reasons in support of unit testing are:

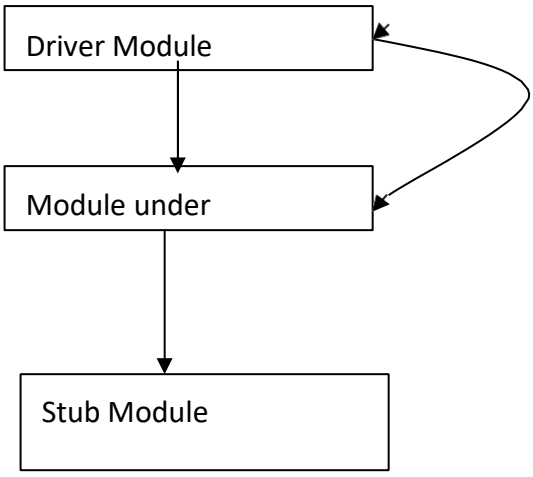
- The size of a single module is small enough that we can locate an error fairlyeasily.
- Confusing interactions of multiple error is widely different parts of the software areeliminated.

Driver and Stub Modules

In order to test a single module, we need a complete environment to provide all that is necessary for execution of the module. We will need the following in order to be able to test the module:

- The procedures belonging to other modules that the module under testcalls.
- Nonlocalb data structures that the moduleaccesses.

Stubs and drivers are design to provide the complete for a module.



Global Data

Fig. 6.2 Unit testing with the help of driver and stub module

A stub procedure is a dummy procedure that has the same I/O parameters as given procedure but has a highly simplified behaviour. A driver module would contain the no local data structure accessed by the module under test, and would also have the code to call the different function of the module with appropriate parameter values.

INTRODUCTION TO BLACK BOX TESTING

There are essentially 3 main approaches for designing test cases for unit testing.

1. black box approach,
2. white box approach

BLACK BOX TESTING

Black Box Testing is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and

specifications.

In the black-box testing, The following are the two main approaches to designing black box test cases.

- Equivalence class portioning
- Boundary value analysis

EQUIVALENCE CLASS PARTITIONING

In this approach, the domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behaviour of the program is similar for every input data belonging to the same equivalence class. The main idea behind defining the equivalence classes is that testing the code with any one value

belonging to an equivalence class is as good as testing the software with any other value.

- Consider percentage field that will accept percentage only between 50 to 90 %, more and even less than not be accepted, and application will redirect user to error page.
- If percentage entered by user is less than 50 %or more than 90 %, that equivalence partitioning method will show an invalid percentage.

BOUNDARY VALUE ANALYSIS

A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs. The reason behind such errors might purely be due to psychological factors. Programmers often fail to see the special processing required by the input values that lie at the boundary of the different equivalence classes.

Summary of the Black-box test suite Design

•Examine the input and output values of the program.

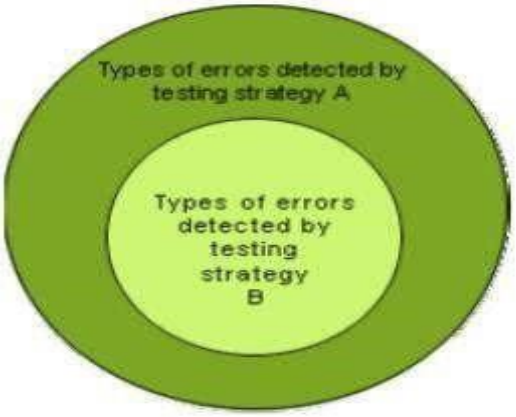
- Identify the equivalence classes.
- Pick the test cases corresponding to equivalence class testing and boundary value

analysis

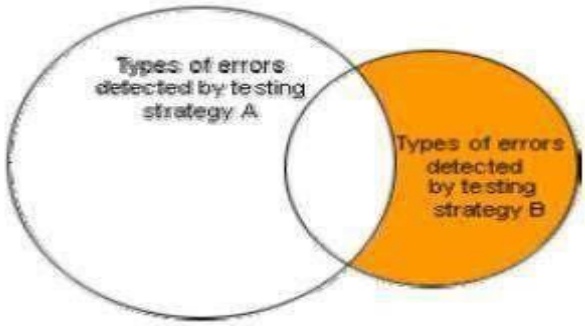
WHITE –BOXTESTING

- White Box Testing is software testing technique in which internal structure, design and coding of software are tested to verify flow of input- output and to improve design, usability and security.
- In white box testing, code is visible to testers so it is also called Clear box testing, Open box testing, Transparent box testing, Code-based testing and Glass box testing.
- One white-box testing strategy is said to be stronger than another strategy, if all types of errors detected by the first testing strategy

is also detected by the second testing strategy, and the second testing strategy additionally detects some more types of errors. The concepts of stronger and complementary testing are schematically illustrated in fig.



A is a stronger testing strategy than B



A and B are complementary testing strategies

DIFFERENT WHITE BOX METHODOLOGIES:

1.STATEMENT COVERAGE

2. BRANCH COVERAGE,

3. CONDITION COVERAGE,

4. PATH COVERAGE,5.DATA FLOW BASED TESTING

6.AND MUTATION TESTING.

STATEMENT COVERAGE

This statement coverage strategy aims to design test cases so that every statement in a program is executed at least once. The principle idea governing the statement coverage strategy is that unless a statement is executed there is no way to determine whether an error exist in that statement unless a statement is executed, we cannot observe whether it causes failure due to some illegal memory access, wrong result

computationetc.

BRANCH COVERAGE

In the branch coverage-based testing strategy, test cases are designed to make each branch condition assume true and false value in turn. Brach testing is also known as edge testing, which is stronger than statement coverage testingapproach.

CONDITION COVERAGE

In this structural testing, test cases are designed to make each component of a composite conditional expression assumes both true and false values. For example, in the conditional expression ((C_1 AND C_2) OR C_3), the components C_1,C_2 andC_3 are each made to assume both true and false values

PATH COVERAGE

The path coverage-based testing strategy requires designing test cases such that all linearly independent paths is the program are executed at least once. A linearly independent path can be defined in the terms of the control flow graph (CFG) of aprogram.

Control Flow Graph (CFG)

A control flow graph describes the sequence in which the different instructions of a program get executed. In other words, a control flow graph describes how the control flows through the program. In order to draw the control flow graph of a program, all

the statements of a program must be numbered first.

PATHlinearly independent path.

PATH

A path through a program is a node and edge sequence from the starting node to a terminal node of the control flow graph of a program.. A program can have more than one terminal nodes when it contains multiple exit or return type of statements.

MCCABE'S CYCLOMATIC COMPLEXITY METRIC

McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program. Also, the McCabe's cyclomatic complexity is very simple to compute. Thus, the McCabe's cyclomatic complexity metric provides a practical way of determining the maximum number of linearly independent

There are three different ways to compute the cyclomatic complexity. The answers computed by the three methods are guaranteed to agree.

DATA FLOW – BASED TESTING

The data flow – based testing method selects the test paths of a program according to the location of the definitions and use of the different variables in aprogram.

Consider a program P. For a statement numbered S of P, let

DEF (S) = {X | Statement S contains a definition of X},

and

USES (S) = {X| Statement S contains a use of X}

For the statement S: a = b+c ; DEF (S) ={ a}, USES(S) ={b,c}

MUTATION TESTING

In mutation testing, the software is first tested by using an initial test suite built of from different white – box testing strategies. After the initial testing is complete, mutation testing is taken up. The idea behind mutation testing is to make a few arbitrary changes to a program at a time. Each time the program is changed, it is called a mutated program and the change effected is called a mutant. A mutated program is tested against the full test suite of the program

DEBUGGING

Once errors are identified, it is necessary to first locate the precise program statements responsible for the errors and then to fix them.

Buffer Force Method

This is the most common method of debugging, but is the least efficient method. In this approach, the program is base with print statement to print the intermediate values with the hope that some of the printed values will help to identify the statement in error.

Backtracking

In this approach, beginning from the statement at which an error symptom is observed, the source code is traced backwards until the error is discovered.

Cause Elimination Method

In this approach, a list of causes which could possibly have contributed to the error symptom is developed and tests are conducted to eliminate each cause.

Program Slicing

This technique is similar to back tracking. However, the search space is reduced by defining slices.

Big – Bang Approach

In this approach, all the modules of the system are simply put together and tested. This technique is practicable only for small systems. The main problem with this approach is that once an error is found during the integration testing.

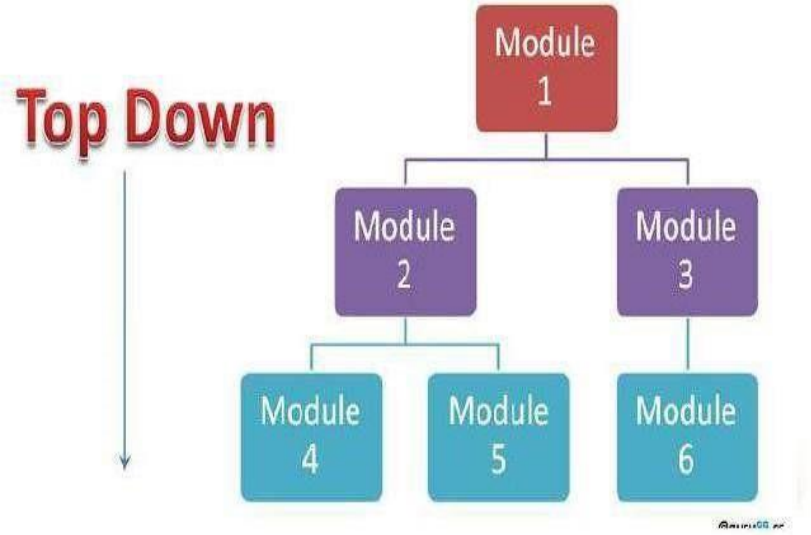
Top – Down Approach

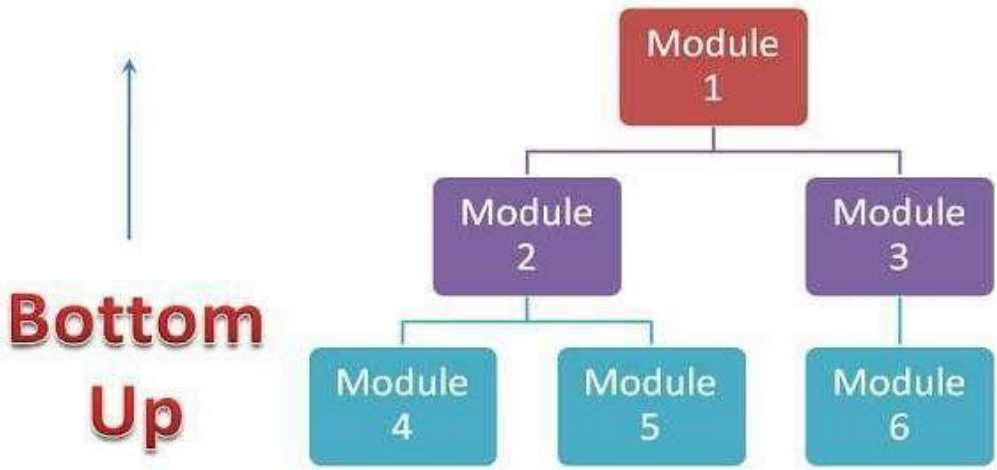
Top-down integration testing starts with the main routine and one or two subordinate routines in the system. After the top-level
'skeleton' has been tested, the immediately subroutines of the 'skeleton'

are combined with it and tested. Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test

Bottom – up Integration Testing

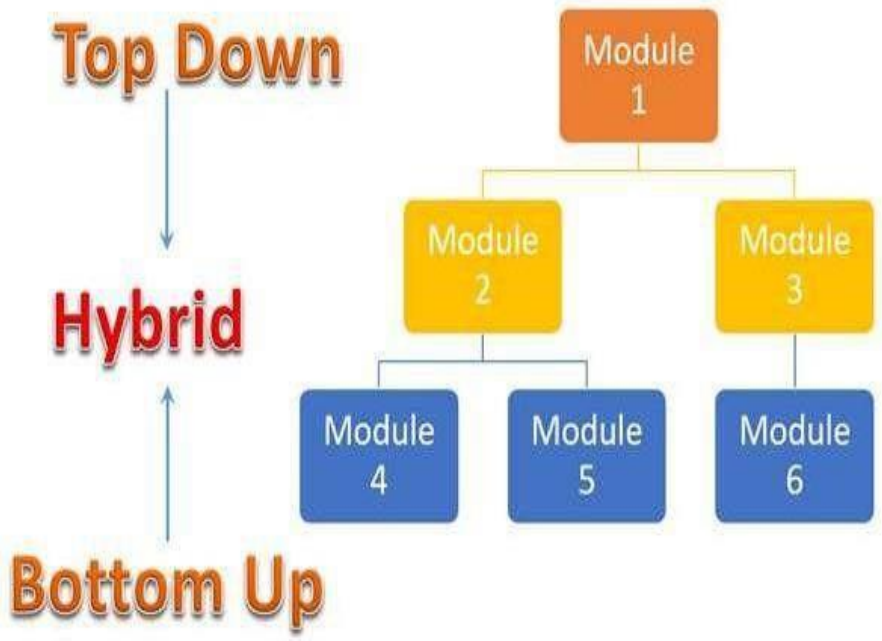
In bottom-up testing, each subsystem is tested separately and then the full system is tested. A subsystem might consist of many modules which communicate among each other through well-defined interfaces. The primary purpose of testing each subsystem is to test the interfaces among various modules making up the subsystem. Both control and data interfaces are tested.





Mixed Integration Testing

A mixed (also called sandwiched) integration testing follows a combination of topdown and bottom-up testing approaches. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready.



Phased vs. incremental testing

The different integration testing strategies are either phased or incremental. A

comparison of these two strategies is as follows:

•In incremental integration testing, only one new module is added to the partial system each time.

System Testing:

System tests are designed to validate a fully developed system to assure that it meets its requirements. Three kinds of system testingare:

- Alphatesting
- Betatesting
- Acceptancetesting

Alpha Testing

Alpha testing refers to the system testing carried out by the team within the developing organization.

Beta testing

Beta testing is the system testing performed by a select group of friendly customers.

Acceptance Testing

Acceptance testing is the system testing performed by the customer to determine whether to accept or reject the delivery of the system.

of thesystem.

Performance Testing

Performance testing is carried out to check whether the system meets thenon –functional requirements identified in the SRS document. The types of

performance testing to be carried out on a system depend on the different nonfunctional requirements of the system document in the SRS document.

All performance tests can be considered as black – boxtests.

- Stress testing
- Volume testing
- Configuration testing
- Compatibility testing

Stress Testing

Stress testing is also known as endurance testing. Stress testing evaluated system performance when it is stressed for short periods of time. Stress tests are black – box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software. Input data volumes, input data rate, processing time, utilization of memory are tested beyond the designedcapacity.

Volume Testing

Volume testing, as the name suggests, is a testing done on high volumes of data. It belongs to a group of non-functional testing that is performed as part of performance-testing where a software product or application with high volume of data is tested,

Configuration Testing

Configuration Testing is the type of Software Testing which verifies the performance of the system under development against various combinations of software and hardware to find out the best configuration under which the system can work without any flaws or issues while matching its functional requirements.

Software

Here, software means different operating systems (Linux, Window, and Mac) and also check the software compatibility on the various versions of the operating systems like Win98, Window 7, Window 10, Vista, Window XP, Window 8, UNIX, Ubuntu, and Mac.

Hardware

The application is compatible with different sizes such as RAM, hard disk, processor, and the graphic card, etc.

Mobile

Check that the application is compatible with mobile platforms such as iOS, Android, etc.

Network

Checking the compatibility of the software in the different network parameters such as operating speed, bandwidth, and capacity.

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

Recovery Testing

Recovery Testing is software testing technique which verifies software's ability to recover from failures like software/hardware crashes, network failures etc. The purpose of Recovery Testing is to determine whether software operations can be continued after disaster or integrity loss. Recovery testing involves reverting back

Maintenance Testing

Most of the tests are conducted on software during its pre-release stage, but some tests are done once the software has been released. One such procedural testing is known as Maintenance Testing.

Documentation Testing

Documentation is checked to ensure that the required user manual, maintenance manuals and technical manuals exist and are consistent.

Usability Testing

Usability Testing also known as User Experience(UX) Testing, is a testing method for measuring how easy and user-friendly a software application is. A small set of target end-users, use software application to expose usability defects. Usability testing mainly focuses on user's ease of using application, flexibility of application to handle controls and ability of application to meet its objectives.

CHAPTER-7

UNDERSTANDING THE IMPORTANCE OF S/W RELIABILITY

DEFINITIONS OF SOFTWARE RELIABILITY Software reliability is defined as

the probability of failure-free operation of a software system for a specified time in a specified environment. The key elements of the definition include probability of failure-free operation, length of time of failure-free operation and the given execution environment

Reliability of a software product essentially denotes its trustworthiness or dependability. Alternatively, reliability of a software product can also be defined as the probability of the product working "correctly" over a given period of time.

- It is obvious that a software product having a large number of defects is unreliable.

Factors Influencing Software Reliability

•A user's perception of the reliability of a software depends upon twocategories of information.

- The number of faults present in the software.
- The way users operate the system. This is known as the operational profile.

Reasons for software reliability being difficult to measure

The reasons why software reliability is difficult to measure can be summarized as follows:

- The reliability improvement due to fixing a single bug depends on where the bug is

located in the code.

- The perceived reliability of a software product is highly observer dependent

HARDWARE RELIABILITY VS. SOFTWARE RELIABILITY

Reliability behavior for hardware and software are very different.

For example, hardware failures are inherently different from software failures. Most hardware failures are due to component wear and tear. A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix hardware faults, one has to either replace or repair the failed part.

- **MEAN TIME TO FAILURE (MTTF).**

MTTF is the average time between two successive failures, observed over a large number of failures. To measure MTTF, we can record the failure data for n failures. Let the failures occur at the time instants t1, t2, …, tn. Then, MTTF can be calculated.

- **MEAN TIME TO REPAIR (MTTR).**

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

- **MEAN TIME BETWEEN FAILURE (MTBR).**

MTTF and MTTR can be combined to get the MTBR metric: MTBF = MTTF + MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours

PROBABILITY OF FAILURE ON DEMAND (POFOD).

Unlike the other metrics discussed, this metric does not explicitly involve

AVAILABILITY

Availability of a system is a measure of how likely shall the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs.

Classification of software failures A possible classification of failures of software products into five different types is as follows:

- Transient. Transient failures occur only for certain input values while invoking a function of the system.
- Permanent. Permanent failures occur for all input values while invoking a function of the system.

interface

SOFTWARE QUALITY

Traditionally, a quality product is defined in terms of its fitness of purpose. That is, a quality product does exactly what the users want it to do. For software products, fitness of purpose is usually interpreted in terms of satisfaction of the requirements laid down in the SRS document. Although "fitness of purpose" is a satisfactory definition of quality for many products such as a car, a table fan, a grinding machine, etc. – for software products,

- **Portability:** A software product is said to be portable, if it can be easily made to work in different operating system environments, in different machines, with other software products, etc.

- **Usability:** A software product has good usability, if different categories of users (i.e., both expert and novice users) can easily invoke the functions of the product.

- **Reusability**: A software product has good reusability, if different modules of the product can easily be reused to develop new products.

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

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

SOFTWARE QUALITY MANAGEMENT

A quality management system (often referred to as quality system) is the principal methodology used by organizations to ensure that the products they develop have the desired quality. A quality system consists of the following:

Managerial Structure and Individual Responsibilities.

A quality system is actually the responsibility of the organization as a whole. However, every organization has a separate quality department to perform several quality system activities.

- review of the quality system
- development of standards, procedures, and guidelines, etc

EVOLUTION OF QUALITY MANAGEMENT SYSTEM

Quality systems have rapidly evolved over the last five decades. Prior to World War II, the usual method to produce quality products was to inspect the finished products to eliminate defective products. Since that time, quality systems of organizations have undergone through four stages of evolution as shown in the fig. The initial product inspection method gave way to quality control (QC