KIIT POLYTECHNIC

LECTURE NOTES

ON

SOFTWARE ENGINEERING

Prepared by

Sunil Kumar Sahoo
Lecturer, Department of Computer Science & Engineering
KIIT Polytechnic, Bhubaneswar
Email Id-sunilfcs@kp.kiit.ac.in

CONTENTS

UNIT-1-INTRODUCTION TO SOFTWARE ENGINEERING

INTRODUCTION TO SOFTWARE:

- Software is a set of instructions used to get inputs and process them to produce the desired output.
- It also includes a set of documents, such as the software manual, Source Code, Executables, Design Documents, Operations, and System Manuals and Installation and Implementation Manuals.

Software Engineering

A few important definitions given by several authors and institutions are as follows:

Definition:

Software Engineering is the application of a systematic, disciplined, scientific approach to the development, operation and maintenance of software.

OR

Software Engineering is a discipline whose aim is the production of fault free software that satisfies the user's needs and that is delivered on time and within budget.

Goals of Software Engineering:

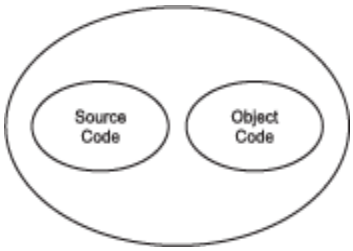
The primary goals of Software Engineering is

1. To improve the quality, reliability of software.
2. To increase productivity.
3. To increase the job satisfaction of software engineers, etc.

PROGRAMS VERSUS SOFTWARE PRODUCTS

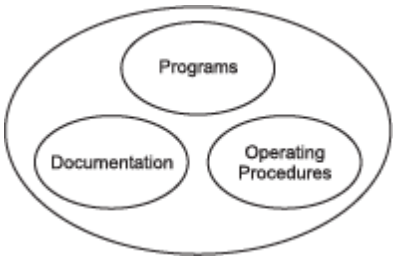
Programs

- A program is a combination of source code and object code.
- A program is a subset of software and it becomes software only if documentation and operating procedure manuals are prepared.
- Program = Source Code + Object Code



Software Products

- A software product consists not only of the program code but also of all the associated documents, such as the requirements specification documents, the design documents, the test documents, and the operating procedures, user manuals and operational manuals, etc.
- Software = Program + Documentation + Operating Procedures



Programs versus Software Products

The various differences between a program product and a software product are given in Table.

S. No.	Programs	Software Products
1.	Programs are developed by individuals for their personal use.	A software product is usually developed by a group of engineers working as a team.
2	Usually small in size	Usually large in size
3	Single user	Large number of users
4	Single developer	Team of developers
5	Lack proper documentation	Good documentation support
6	Adhoc development	Systematic development
7	Lack of user interface	Good user interface
8	Have limited functionality	Exhibit more functionality

SOFTWARE-DEVELOPMENT LIFE-CYCLE

- The software-development life-cycle is used to help the development of a large software product in a systematic, well-defined, and cost-effective way.
- It goes through a series of phases from start to end. This process is called the Software-Development Life-Cycle.
- The software development life-cycle consists of several phases.
- These are:
  - Feasibility study
  - Project analysis (Requirements analysis and specification)
  - System design (Design)
  - Coding
  - Testing
  - Implementation
  - Maintenance

Software Development Life Cycle Models

- A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle.

Need for a software life cycle Model:

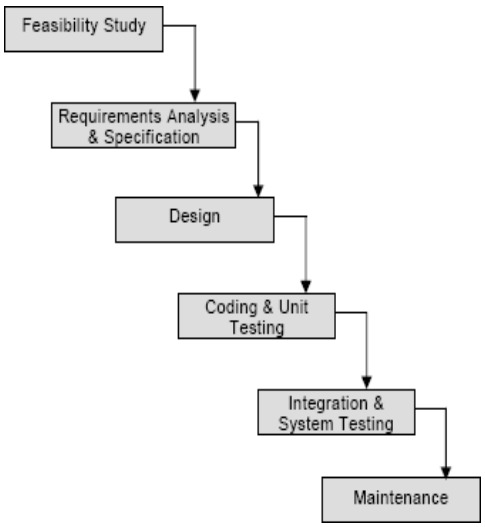
- The development team must identify a suitable life cycle model to develop a project..
- Without using of a particular life cycle model the development of a software product may lead to project failure.

Different software life cycle models:

- Various kinds of process models:
  - Classical Waterfall model
  - Iterative Waterfall model
  - Prototyping model
  - Incremental model
  - Spiral model

Classical waterfall model:

- The waterfall model is a very common and traditional SDLC model or software process model.
- The figure is given below.
- This model is known as the *waterfall model*, because of the cascade from one phase to another phase.
- As the figure shows, the phases are cascade in nature, where the output of one phase is the input to the next one. Each phase performs a set of activities.
- All other life cycle models are essentiallyderived from the classical waterfall model.
- Classical waterfall model has the following phases.
    - Feasibility Study
    - Requirements Analysis and Specification
    - Design
    - Coding and Unit Testing
    - Integration and System Testing, and
    - Maintenance



Feasibility study:

The main aim of feasibility study is to determine whether the project is ***financially and technically feasible or not,*** to develop the product.

- At first project managers or team leaders try to have a rough understanding of the project by visiting the client side. They study or discuss different aspects of project like resources required, cost, time for development etc.
- At the end of this phase, a report called a feasibility study is prepared by a group of software engineers.

Requirements analysis and specification:

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

- Requirements gathering and analysis, and
- Requirements specification.

- The goal of the requirements gathering activity is to collect all the related information from the customer regarding the product to be developed.
- The final document is known as the software requirement specification (SRS) document.

Design:

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation by using some programming language.

- Two distinctly different approaches are available:
  - Architectural Design approach.
  - Detailed Design approach.
- **Architectural Design:**
  - It involves:
    - Identifying the software components.
    - Decomposing the software components into modules.
    - Specifying the interconnection between the various components.
- **Detailed Design:**
  - It is deals with all details for the implementation i.e. procedures to process the algorithms, data structures to be used, etc.
- **Output:**
  - Design Document
  - Test Plan

Coding and unit testing:

- The purpose of the coding and unit testing phase (sometimes called the implementation phase) of software development is to translate the software design into source code.
- The end of this phase, each unit module is tested against each unit test plan.
- Output of this phase is:
  - Program code or source code
  - Unit test report

Integration and System Testing:

- Integration of different modules is undertaken once they have been coded and unit tested.
- During the integration and system testing phase, the modules are integrated.
- System testing usually consists of three different kinds of testing activities:-
  - **α – testing:** It is the system testing performed by the development team.
  - **β – Testing:** It is the system testing performed by a friendly set of customers.
  - **Acceptance testing:** It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.
- Output:
  - Final test report
  - Total product

Maintenance:

- The product is ready and it is delivered to the client.
- Maintenance of a typical software product requires much more than the effort necessary to develop the product itself.
- The relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratio.
- Maintenance involves performing any one or more of the following three kinds of activities:
  - ***corrective maintenance***
  - ***perfective maintenance***
  - ***adaptive maintenance***

Advantages of Waterfall Model

- It is a linear model.
- Relatively easy to understand and manage.
- It is systematic and sequential.
- It is a simple one.
- It has proper documentation.

Disadvantages of waterfall model:

- It is difficult to define all requirements at the beginning of a project.
- This model is not suitable for accommodating any change.
- It does not suitable to large projects.
- It involves heavy documentation.
- We cannot go backward in the SDLC.
- There is no risk analysis.

What is maintenance? Different types of maintenance. Or short note on maintenance.

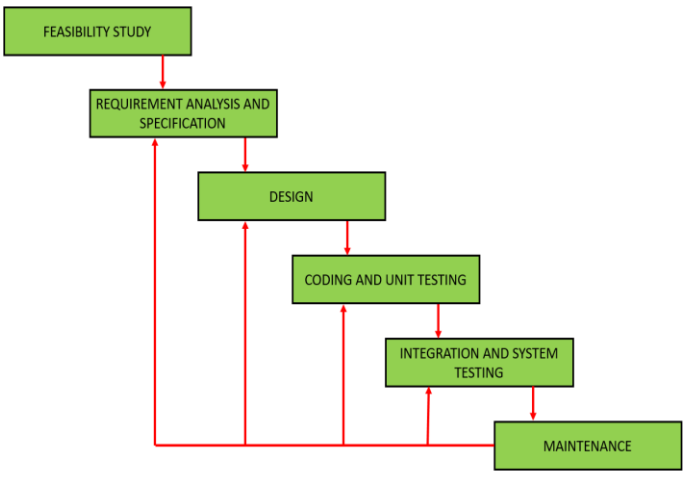
Maintenance:

- The product is ready and it is delivered to the client.

- Maintenance of a typical software product requires much more than the effort necessary to develop the product itself.
- The relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratio.
- Maintenance involves performing any one or more of the following three kinds of activities:
    - ***corrective maintenance***
    - ***perfective maintenance***
    - ***adaptive maintenance***
    - Correcting errors that were not discovered during the product development phase. This is called ***corrective maintenance***.
    - Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements. This is called ***perfective maintenance.***
    - Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. This is called ***adaptive maintenance***.

Iterative Waterfall Model:

- Beside the weaknesses of the waterfall model, **Iterative development** is the heart of acyclic software development process.
- It starts with feasibility study and ends with deployment (maintenance) with the cyclic interactions in between phases. i.e. it provide feedback paths from every phase to its preceding phases.
- The feedback paths allow for correction of the errors done during a phase.



- The principle of detecting errors as close to its point of introduction as possible. This is known as "Phase Containment of Errors".
- To achieve this review is done after completion of each phase.
- The rest of the phases of development are same as classical water fall model. The model is pictorially shown in fig.

Prototyping Model:

It sometimes happens that a customer defines a set of general objectives for software but does not identify detailed requirements (like input, processing, or output).

Or in other cases, the developer may not be sure about the efficiency of an algorithm, operating system, used etc. In these situations, a prototyping model may be used.

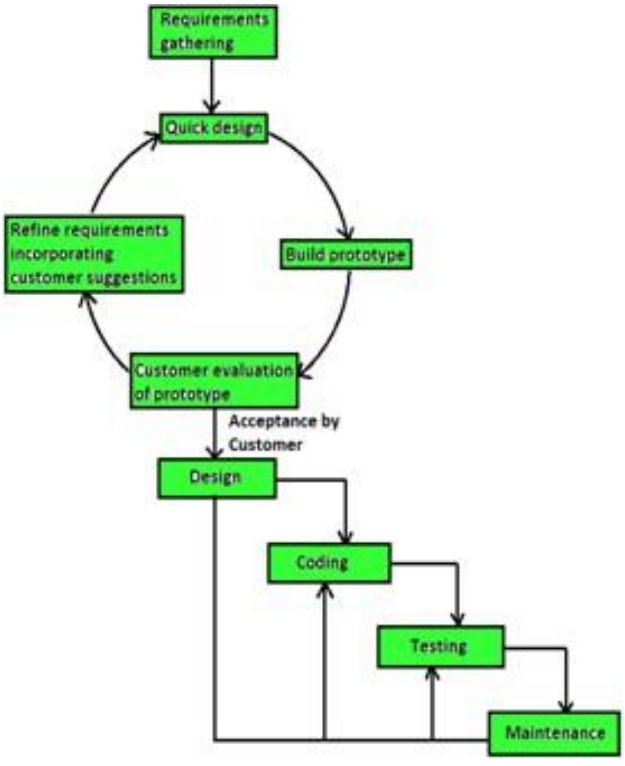
- A prototype is a toy implementation of the system.
- A prototype usually shows limited functional capabilities, low reliability, and inefficient performance.
- It uses dummy functions.
- A prototype is the partial version of the actual system.

Where to use Prototype Model:

A prototype of the actual product is preferred in situations such as:

- User requirements are not complete.
- Technical issues are not clear.

The prototyping model of software is shown in fig:



The phases of this model are:

Requirement Gathering:

- In this model a prototyping starts with an initial requirement and gathering phase.
- The functionalities required for the product are gathered.

Quick Design:

- A quick design is carried out and a prototype is build.
- A new plan is created from gathered requirements for the prototype to be built.

Prototype creation:

- Based on the design issues the trial product is created.

Customer Evaluation:

- The developed prototype is submitted to the customer for evaluation.
- Based on the customer feedback we may go for development or again go for the redesign as per customer demand.

Prototype refinement:

- Based on the information.

The code for the prototype is usually thrown away, but the experience gained while developing the prototype helps for actual development. The rest of the phases for development are similar to the waterfall model.

Advantages of Prototyping Models:

- Suitable for large systems.
- Customer communication is available.
- Quality of software is good.
- It helps to identifying the requirements.

Limitations of Prototyping Model:

- It is very difficult to predict how the system will work after development.
- This model is time consuming.

Incremental Model or Evolutionary Model:

This life cycle model is also referred to as the successive versions model and sometimes as the **incremental model**.

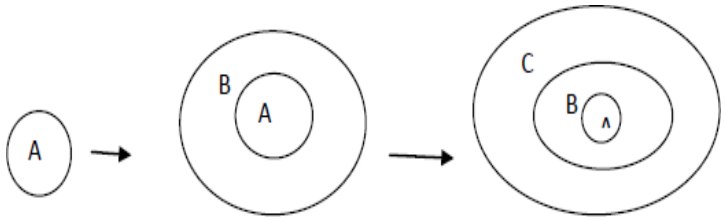


Fig: Evolutionary model (A,B,C are modules of the software model)

- In this lifecycle model the software is first broken down into several modules which can be incrementally constructed and delivered.
- The development team first develops the core modules of the system.
- The development model is shown in fig.
- Each successive version of the product is fully functioning capable of performing more useful work than the previous version.
- In this model the user gets a chance to experiment with particularly developed software much before the complete version of the software is ready.
- This model is useful only for very large products.

Advantages:

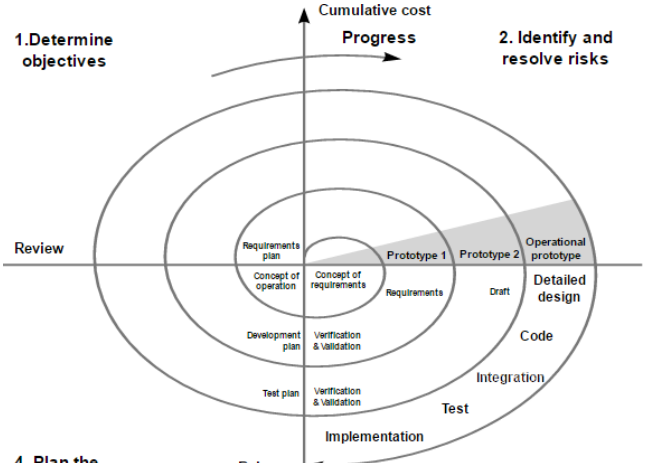
- Problem understanding increases through successive refinements
- Performs cost benefit analysis before enhancing software with capabilities
- Does not involve high complexity rate
- Early feedback is generated

Disadvantages

- Requires planning at the management and technical level.
- Becomes invalid when there is time constraint in the project schedule.

Spiral Model

- The spiral model, originally proposed by Barry Boehm.
- Main objective is to analyze the risks involved in the software project.
- Combines the features of prototyping model, evolutionary and iterative waterfall model. Hence this model is also called meta model.
- The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed.
- Each loop of the spiral represents a phase



of the software process. For example, the innermost loop might be feasibility study, the next loop may be requirements specification, and so on.

– Each phase in this model is split into four sectors (or quadrants) as shown in fig.The following activities are carried out during each phase of a spiral model.

First quadrant (Objective Setting)

- During the first quadrant, it determines objectives, alternatives &constraints.

Second Quadrant (Risk Assessment and Reduction)

- A detailed analysis is carried out for each identified project risk.
- Steps are taken to reduce the risks.

Third Quadrant (Development and Validation)

- Develop and validate the next level of the product after resolving the identified risks.

Fourth Quadrant (Review and Planning)

- Review the results achieved so far with the customer and plan the next iteration around the spiral.
- Progressively more complete version of the software gets built with each iteration around the spiral.

Advantages

– For large, complex and expensive projects.
– If the software associated with risk.
– Re-evaluation after each step

Disadvantages

– Assessment and resolution of project risks is not an easy task.
– Difficult to estimate budget and schedule in the beginning.

UNIT-2
Project Planning and Project Estimation Techniques

Responsibilities of a software project manager

- Software project managers take the overall responsibility of a project to success.
- It is very difficult to particularly describe the job responsibilities of a project manager.
- The job responsibility of a project manager ranges from building up team morale to highly visible customer presentations.
- Most managers take responsibility for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, interfacing with clients, managerial report writing and presentations, etc.

Skills necessary for software project management

- A theoretical knowledge of different project management techniques is certainly necessary to become a successful project manager.
- However, effective software project management frequently calls for good qualitative judgment and decision taking capabilities.
- Should have good knowledge about of latest software.

Project planning

Once a project is found to be feasible, software project managers undertake project planning. Project planning is undertaken and completed before any development activity starts. Project planning consists of the following essential activities:

• Estimating the following attributes of the project:

Project size: What will be problem complexity in terms of the effort and time required to develop the product?
Cost: How much is it going to cost to develop the project?
Duration: How long is it going to take to complete development?
Effort: How much effort would be required?

Software Project Management Plan (SPMP)

- Once project planning is complete, project managers document their plans in a Software Project Management Plan (SPMP) document.
- The SPMP document should discuss a list of different items that have been discussed below.

1. **Introduction** (Objectives, Major Functions, Performance Issues, Management and Technical Constraints).
2. **Project Estimates (**Historical Data Used, Estimation Techniques Used, Effort, Resource, Cost, and Project Duration Estimates).
3. **Schedule (**Work Breakdown Structure, Activity Network Representation, Gantt Chart Representation, PERT Chart Representation).
4. **Project Resources (**People, Hardware and Software, Special Resources).
5. **Staff Organization**
6. **Risk Management Plan (**Risk Analysis, Risk Identification, Risk Estimation)
7. **Project Tracking and Control Plan**
8. **Miscellaneous Plans (**Quality Assurance Plan, Configuration Management Plan**)**

Sliding Window Planning

- Project planning is a very difficult for large projects to make accurate plans.
- Project planning requires utmost care and attention otherwise projects will failure.
- Schedule delays can cause customer dissatisfaction and affect the team morale.

- This happens due to the fact that the proper parameters, scope of the project, project staff, etc. may change during the span of the project.
- In order to overcome this problem, sometimes project managers undertake project planning in stages. This protects managers from making big commitments too early.
- This technique is known as Sliding Window Planning. In the sliding window technique, starting with an initial plan, then go for successive development stages.
- At the start of a project, project managers have incomplete knowledge about the details of the project. Their information base gradually improves as the project progresses through different phases.
- After the completion of every phase, the project managers can plan each subsequent phase more accurately and with increasing levels of confidence.

Metrics for software project size estimation

- In order to be able to accurately estimate the project size, some important metrics should be defined in terms of which the project size can be expressed.
- Currently two metrics are popularly being used widely to estimate size:
  - **Lines of Code (LOC)**
  - **Function Point (FP)**
- The usage of each of these metrics in project size estimation has its own advantages and disadvantages.

Lines of Code (LOC)

- LOC is the simplest, very popular among all metrics available to estimate project size.
- Using this metric, the project size is estimated by counting the number of source instructions/lines in the developed program.
  - While counting the number of source instructions, lines used for commenting the code and the header lines should be ignored.
- Determining the LOC count at the end of a project is a very simple job. But, at the beginning of a project is very difficult to accurate measure the LOC count.
- In order to estimate the LOC count at the beginning of a project, project managers usually divide the problem into modules and each module into sub modules and so on, until the sizes of the different leaf-level modules can be approximately predicted.
- To be able to do this, past experience in developing similar products is helpful.

Disadvantages of Using LOC

- Size can vary with coding style.
- Focuses on coding activity alone.
- Measures lexical / textual complexity only.
  - Does not address the issues of structural or logical complexity.
- Difficult to estimate LOC from problem description.

Function point (FP)

- Function point metric was proposed by Albrecht [1983].
- This metric overcomes many of the shortcomings of the LOC metric.
- By using the function point metric we can easily estimate the size of a software product directly from the problem specification. .
- This is in contrast to the LOC metric, where the size can be accurately determined only after the product has fully been developed.
- The FP can be computed in two steps:
  - Unadjusted Function Point (UFP)
  - Technical complexity Factor (TFC)
- Once the UFP is computed, the TFC is computed next.

FP=UFP+TFC.

Project Estimation techniques

- The basic project planning activity is Estimation of various project parameters.
- The important project parameters are: project size, effort required to develop the software, project duration, and cost.
- These estimates not only help in quoting the project cost to the customer, but are also useful in resource planning and scheduling.
- There are three broad categories of estimation techniques:
  - **Empirical estimation techniques**
  - **Heuristic techniques**
  - **Analytical estimation techniques**

Empirical Estimation Techniques:

- This techniques based on making an educated guess of the project parameters.
- While using this technique, prior experience with similar products is helpful.
- Two popular empirical estimation techniques are:
  - Expert judgment technique
  - Delphi cost estimation.

Expert Judgment Technique

- It is one of the most widely used estimation techniques.
- In this approach, an expert makes an educated guess of the problem size after analyzing the problem thoroughly.
- Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) of the system and then combines them to arrive at the overall estimate.

Delphi cost estimation

- This approach tries to overcome some of the shortcomings of the expert judgment approach.
- It is carried out by a team comprising of a group of experts and a coordinator.
- In this approach, the coordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate.
- Estimators complete their individual estimates anonymously and submit to the coordinator.
- In their estimates, the estimators mention any unusual characteristic of the product which has influenced his estimation.
- The coordinator prepares and distributes the summary of the responses of all the estimators, and includes any unusual rationale noted by any of the estimators. Based on this summary, the estimators re-estimate.
- This process is iterated for several rounds.
- However, no discussion among the estimators is allowed during the entire estimation process.
- The idea behind this is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior.
- After the completion of several iterations of estimations, the coordinator takes the responsibility of compiling the results and preparing the final estimate.

Heuristic Techniques:

- This technique assumes that the relationships among the different project parameters can be modeled using suitable mathematical expressions.
- Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the value of the basic parameters in the mathematical expression.
- Different heuristic estimation models can be divided into the following two classes:
  - single variable model and
  - The multi variable model.

COCOMO Model:

- The Constructive Cost Model (COCOMO) is an heuristic estimation model i.e., the model uses a theoretically derived formula to predict cost related factors. This model was created by "Barry Boehm". The COCOMO model consists of three models:-
- The COCOMO models are defined for three classes of software projects, stated as follows i.e. it divides software product developments into 3 categories:
  - **Organic**
  - **Semidetached**
  - **Embedded**

Organic:

- A development project can be considered of organic type, if
  - The project deals with developing a well understood application program.
  - The size of the development team is reasonably small.
  - The team members are experienced in developing similar types of projects.

Semidetached:

- A development project can be considered of semidetached type, if
  - The development consists of a mixture of experienced and inexperienced staff.
  - Team members may have limited experience on related System.

Embedded:

- A development project is considered to be of embedded type, if
  - The software being developed is strongly coupled to complex hardware.

- The COCOMO model consists of three models(i.e. Software cost estimation is done through three stages):
  - **Basic COCOMO,**
  - **Intermediate COCOMO,**
  - **Complete COCOMO.**

Basic COCOMO Model

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

  $\text{Effort} = a_1 * (\text{KLOC})^{a_2}\ \text{PM}$

  $T_{dev} = b_1 * (\text{Effort})^{b_2}\ \text{Months}$

- Where
  - KLOC is the estimated size of the software product expressed in Kilo Lines of Code,
  - a_1, a_2, b_1, b_2 are constants for each category of software products,
  - T_{dev} is the estimated time to develop the software, expressed in months,
  - Effort is the total effort required to develop the software product, expressed in person months (PMs).

- **Estimation of development effort:**

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

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

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

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

- **Estimation of development time:**

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

  Organic : T_{dev} **= 2.5(Effort)0.38 Months**

  Semi-detached : T_{dev} **= 2.5(Effort)0.35 Months**

  Embedded : T_{dev} **= 2.5(Effort)0.32 Months**

Intermediate COCOMO model:

- The basic COCOMO model assumes that effort and development time are functions of the product size alone.
  - However, a host of other project parameters besides the product size affect the effort required to develop the product as well as the development time.
- Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account.
  - This model recognizes this fact and refines the initial estimate obtained using the basic COCOMO expressions by using a set of 15 cost drivers (multipliers) based on various attributes of software development.
  - Boehm requires the project manager to rate these 15 different parameters for a particular project on a scale of one to three.

- **Example:**
  - If modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1.
  - If there are reliability requirements on the software product, this initial estimate is scaled upward.

- In general, the cost drivers can be classified as being attributes of the following items:

Product:

- It include complexity of the product, reliability requirements of the product, etc.

Computer:

- It include the execution speed required, storage space required etc.

Personnel:

- It includes the experience level of personnel, programming capability, analysis capability, etc.

Development Environment:

- Development environment attributes capture the development facilities available to the developers.
- Ex: automation (CASE) tools.

Complete COCOMO model:

- A major shortcoming of both the basic and intermediate COCOMO models is that they consider a software product as a single homogeneous entity.
- However, most large systems are made up several smaller sub-systems. These sub-systems may have widely different characteristics.
- For example, some sub-systems may be considered as organic type, some semidetached, and some embedded.
- Not only that the inherent development complexity of the subsystems

Analytical Estimation Techniques

- It derive the required results starting with basic assumptions regarding the project.
- Thus, unlike empirical and heuristic techniques, analytical techniques do have scientific basis. Halstead's software science is an example of an analytical technique.
- Halstead's software science can be used to derive some interesting results starting with a few simple assumptions.

Halstead's Software Science – An Analytical Technique:

- It 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 overall program length, potential minimum value, actual volume, effort, and development time.
- For a given program, let:
    - η_1 be the number of unique operators used in the program,
    - η_2 be the number of unique operands used in the program,
    - N_1 be the total number of operators used in the program,
    - N_2 be the total number of operands used in the program.
- It derived expressions for:
    - over all program length,
    - potential minimum volume
    - actual volume,
    - language level,
    - effort, and
    - development time.
    - Staffing level estimation:
- Once the effort required to develop a software has been determined, it is necessary to determine the staffing requirement for the project.
- Putnam first studied the problem of what should be a proper staffing pattern for software projects. He extended the work of Norden who had earlier investigated the staffing pattern of research and development (R&D) type of projects.

Putnam's Work

- Putnam studied the problem of staffing of software projects and found that the software development has characteristics very similar to other R & D projects studied by Norden and that the Rayleigh-Norden curve can be used to relate the number of delivered lines of code to the effort and the time required to develop the project.
- By analyzing a large number of army projects, Putnam derived the following expression:

$$L = C_k K^{1/3} t_d^{4/3}$$

$$K = C/t_d^4$$

For the same product size, $C = L^3 / C_k^3$ is a constant.

or, $K_1/K_2 = t_{d2}^4/t_{d1}^4$

or, $K \propto 1/t_d^4$

or, $cost \propto 1/t_d$

(as project development effort is equally proportional to project development cost)

The various terms of this expression are as follows:

- K is the total effort expended (in PM) in the product development and L is the product size in KLOC.
- t_d corresponds to the time of system and integration testing. Therefore, t_d can be approximately considered as the time required developing the software.

Effect of schedule change on cost

From the above expression, it can be easily observed that when the schedule of a project is compressed, the required development effort as well as project development cost increases in proportion to the fourth power of the degree of compression. It means that a relatively small compression in delivery schedule can result in substantial penalty of human effort as well as development cost. For example, if the estimated development time is 1 year, then in order to develop the product in 6 months, the total effort required to develop the product (and hence the project cost) increases 16 times.

Jensen Model

- Jensen model is very similar to Putnam model.
  - attempts to soften the effect of schedule compression on effort
  - makes it applicable to smaller and medium sized projects.
- Jensen proposed the equation:
  - L=CtetdK1/2
  - Where,
    - Cte is the effective technology constant,
    - td is the time to develop the software, and

K is the effort needed to develop the software

Project scheduling:

Project-task scheduling is an important project planning activity. It involves deciding which tasks would be taken up when. In order to schedule the project activities, a software project manager needs to do the following:

1. Identify all the tasks needed to complete the project.
2. Break down large tasks into small activities.

Page

3. Determine the dependency among different activities.
4. Establish the most likely estimates for the time durations necessary to complete the activities.
5. Allocate resources to activities.
6. Plan the starting and ending dates for various activities.
7. Determine the critical path. A critical path is the chain of activities that determines the duration of the project

Work breakdown structure

- WBS is used to decompose a given task set recursively into small activities.
- It provides a notation for representing the major tasks need to be carried out in order to solve a problem.
- The root of the tree is labeled by the problem name.
- Each node of the tree is broken down into smaller activities that are made the children of the node.
- Each activity is recursively decomposed into smaller sub-activities until at the leaf level, the activities requires approximately two weeks developing.

Activity networks

- WBS representation of a project is transformed into an activity network by representing activities identified in WBS along with their interdependencies.
- An activity network shows the different activities making up a project, their estimated durations, and interdependencies.
  - Each activity is represented by a rectangular node and the duration of the activity is shown alongside each task.

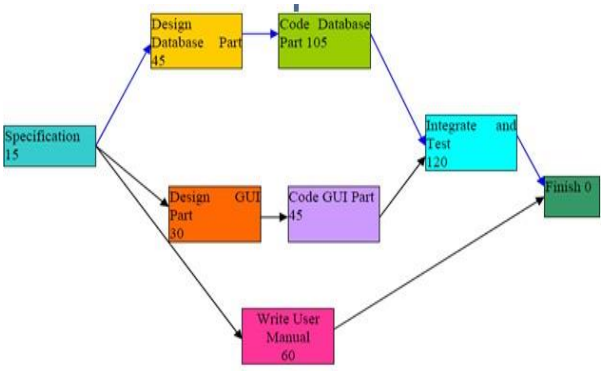


Fig. : Activity network representation of the MIS problem

Gantt chart

- Used to allocate resources to activities.
  - The resources allocated to activities include staff, hardware, and software.
- Gantt charts (named after its developer Henry Gantt) are useful for resource planning.
- It is a special type of bar chart where each bar represents an activity.
  - The bars are drawn along a time line.
  - The length of each bar is proportional to the duration of time planned for the corresponding activity.

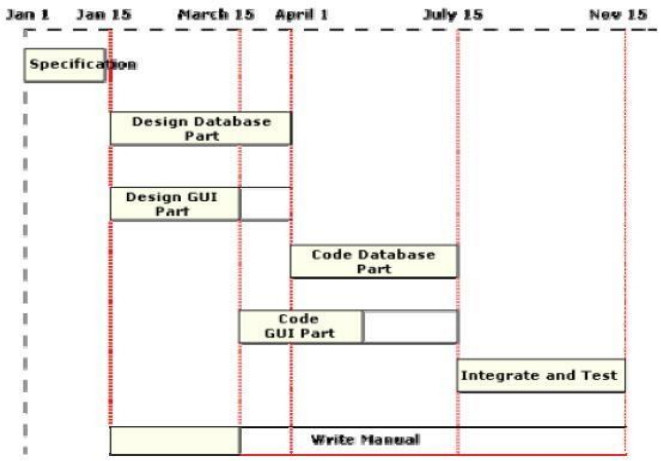


Fig. Gantt chart representation of the MIS problem

PERT chart

- PERT (Project Evaluation and Review Technique) charts consist of a network of boxes and arrows.
  - The boxes represent activities and the arrows represent task dependencies.
- It represents the statistical variations in the project estimates.
  - Thus, in a PERT chart instead of making a single estimate for each task, pessimistic, likely, and optimistic estimates are made.

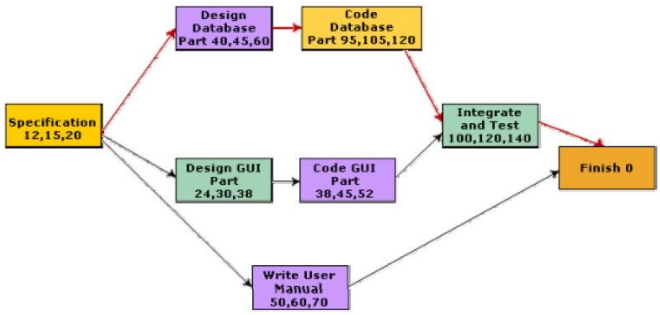


Fig. PERT chart representation of the MIS problem

- The boxes of PERT charts are usually annotated with the pessimistic, likely, and optimistic estimates for every task. Since all possible completion times between the minimum and maximum duration for every task has to be considered.

Project Organization

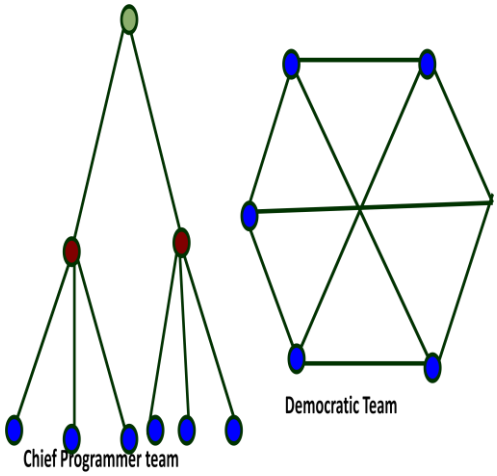
- Engineers get assigned to a project for the entire duration of the project
  - Same set of engineers carry out all the phases
- Advantages:
  - Engineers save time on learning details of every project.
  - Leads to job rotation

Team Structure

- Problems of different complexities and sizes require different team structures:
  - Chief-programmer team
  - Democratic team
  - Mixed organization

Democratic Teams

- Suitable for:
  - small projects requiring less than five or six engineers
  - research-oriented projects
- A manager provides administrative leadership:
  - at different times different members of the group provide technical leadership.
- Democratic organization provides
  - higher morale and job satisfaction to the engineers
  - Suitable for less understood problems,
  - a group of engineers can invent better solutions than a single individual.
- Disadvantage:
  - team members may waste a lot time arguing about trivial points:
    - absence of any authority in the team.

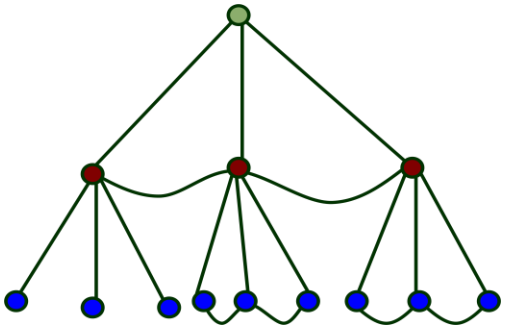


Chief Programmer Team

- A senior engineer provides technical leadership:
  - partitions the task among the team members.
  - verifies and integrates the products developed by the members.
- Works well when
  - the task is well understood
    - also within the intellectual grasp of a single individual,
  - importance of early completion outweighs other factors
    - team morale, personal development, etc.
- Chief programmer team is subject to single point failure:
  - too much responsibility and authority is assigned to the chief programmer.

Mixed Control Team Organization

- Draws upon ideas from both:
  - democratic organization and
  - chief-programmer team organization.
- Suitable for large organizations.



UNIT-3-Requirement Analysis and Specification

- **Requirements gathering and analysis**
- **Software Requirements Specification**
  - **Contents of SRS**
  - **Characteristics of Good SRS**
  - **Organization of SRS**
  - **Techniques for Represents Complex Logic**

Introduction To Requirement Engineering:

A requirement is a feature of the system or a description of something the system is capable of doing in order to fulfill the system's purpose.

— Requirements describe the "what" of a system, not the "how."

— Requirements engineering produces one large document, written in a natural language, and contains a description of what the system will do without describing how it will do it.

— It can be defined as a discipline, which addresses requirements of objects all along a system-development process.

Types of Requirements or Content of SRS Document:

— There are various categories of the requirements. On the basis of their functionality, the requirements are classified into the following two types:

Functional requirements:

- It describes what the software has to do. They are often called product features.
- They define factors, such as I/O formats, storage structure, computational capabilities, timing, and synchronization.

Non-functional requirements:

- These are mostly quality requirements. That specify how well the software does, what it has to do.
- They define the properties or qualities of a product including usability, efficiency, performance, space, reliability, portability, etc.

Constraints on the System:

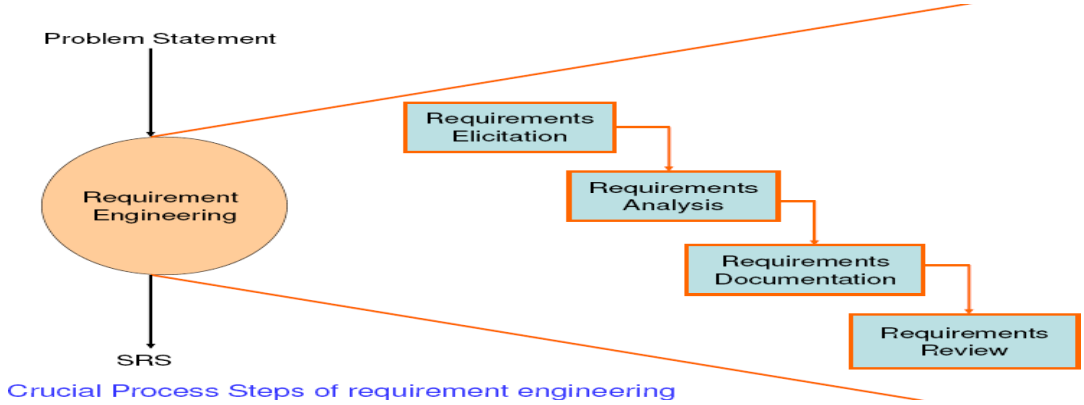
- It describes certain things that the system should or should not do.
- For example, constraints on a function can describe how fast the system can produce results so that it does not overload another system to which it supplies data, etc.
- Examples: Hardware to be used, Operating system or DBMS to be used, Capabilities of I/O devices, Standards compliance, etc.

PROCESS OF REQUIREMENTS ENGINEERING:

— Requirements engineering consists of the following processes:

- Requirements gathering (elicitation).
- Requirements analysis.
- Requirements documentation.
- Requirements review.

— Figure the process steps of requirements engineering.



Crucial Process Steps of requirement engineering

Requirement Elicitation/ Gathering:

- Requirement gathering is a communication process between the parties.
- The requirements are gathered from various sources. Customer (Initiator)
  - End Users
  - Primary Users
  - Secondary Users
  - Stakeholders
- The tools in elicitation are meetings, interviews, video conferencing, e-mails, and existing documents study and facts findings.

Requirement Analysis:

- Requirement analysis is a very important and essential activity after elicitation.
- It analyze, refine and scrutinize requirements to make consistent & unambiguous requirements.
- In this phase, each requirement is analyzed or identified of validity, consistency, and feasibility, etc.

Requirements Documentation:

- Requirements documentation is a very important activity, which is written after the requirements elicitation and analysis.
- This is the way to represent requirements in a consistent format.
- The requirements document is called the Software Requirements Specification (SRS).

Requirements Review:

- A requirements review is a manual process. It is used to improve the quality of SRS.
- This is also called Requirement Verification.
- A requirements review can be informal or formal.
- It should be treated as continuous activity that is incorporate into elicitation, analysis and documentation.

Characteristics of Good SRS:

- Software requirement specification (SRS) is a document that completely describes what the proposed software should do without describing how software will do it.
- The basic goal of the requirement phase is to produce the SRS, Which describes the complete behavior of the proposed software.
- The following are the good characteristic of good SRS:

1. Correct
2. Complete
3. Unambiguous
4. Verifiable
5. Consistent

6. Modifiable
7. Traceable

Correct: SRS is correct when all the requirements of user are stated in the requirements document. Correctness ensures that all specified requirements are performed correctly.

Unambiguous: SRS is unambiguous when every stated requirement has only one interpretation. This implies that each requirement is uniquely interpreted.

Complete: SRS is complete when the requirements clearly define what the software is required to do. I.e. completeness ensures that everything should definitely be specified.

Modifiable: The requirements of the user can change, hence requirements document should be created in such a manner that those changes can be modified easily.

Traceable: SRS is traceable when the source of each requirement is clear and in future it is easier to refer each requirement.

Verifiable: An SRS is verifiable if and only if every stated/mentioned requirement is verifiable. But there should exists some cost-effective process/tools that can check whether the final software meets that requirement.

Consistent: An SRS is consistent if there is no requirement that conflicts with another. For example, there can be a case when different requirements can use different terms to refer to the same object. For instance, a requirement states that an event 'a' is to occur before another event 'b'. But then another set of requirements states that event 'b' should occur before event 'a'.

Organization of the SRS Document:

- Organization of the SRS document 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 (constraints).
- The SRS document should be organized into:

1. Introduction
   (a) Background
   (b) Overall Description
   (c) Environmental Characteristics
   (i) Hardware (ii) Peripherals (iii) People
1. Goals of implementation
   Functional requirements
   Nonfunctional Requirements
   Behavioural Description
   (a) System States
   (b)Events and Actions

The `introduction' section describes:

- The context/ background in which the system is being developed.
- An overall description of the system
- Environmental characteristics.
  - The environmental characteristics subsection describes the properties of the environment with which the system will interact.

Techniques for Represents Complex Logic:

Decision tree

- A decision tree gives a graphic view of the processing logic and the corresponding actions taken.
- The edges of a decision tree represent conditions and the leaf nodes represent the actions to be performed.

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.

Action: If the membership is valid then membership expiry date is updated and the annual membership bill is printed, otherwise an error message is displayed.

Cancel membership option-

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

Action: The membership is cancelled, a cheque for the balance amount due to the member is printed and finally the membership record is deleted from the database.

Decision tree representation of the above example -

The following tree shows the graphical representation of the above example. After getting information from the user, the system makes a decision and then performs the corresponding actions.

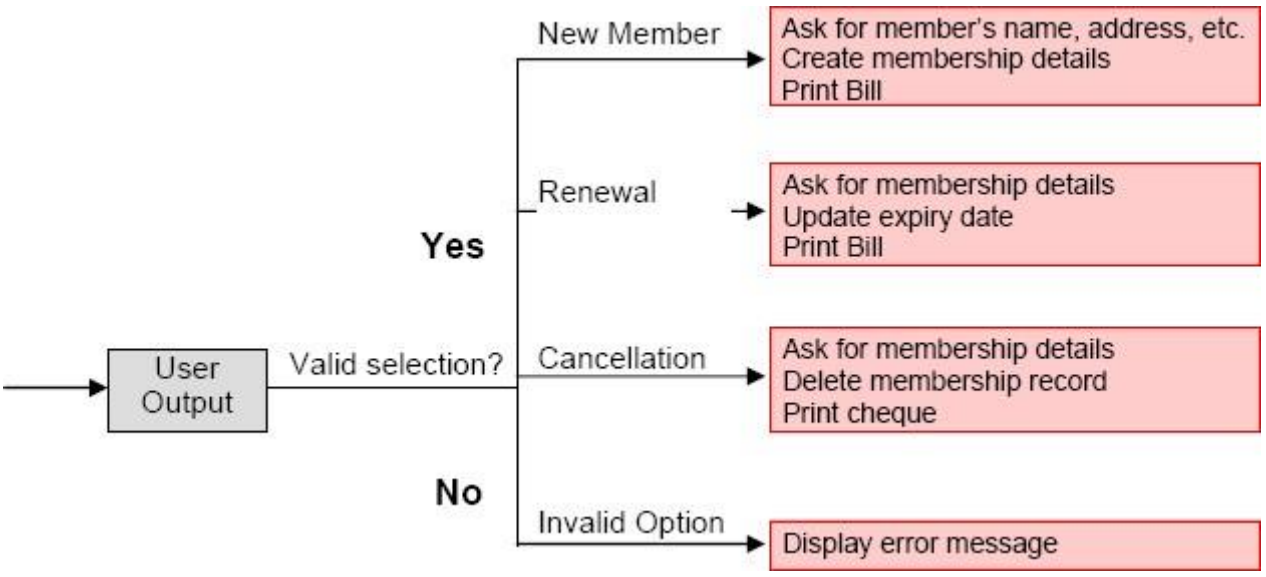


Fig. Decision tree for LMS

Decision table

- A decision table is used to represent the complex processing logic in a tabular or a matrix form.
- The upper rows of the table specify the variables or conditions to be evaluated.
- The lower rows of the table specify the actions to be taken when the corresponding conditions are satisfied.
- A column in a table is called a *rule*. A rule implies that if a condition is true, then the corresponding action is to be executed.

Example: -

Consider the previously discussed LMS example. The following decision table shows how to represent the LMS problem in a tabular form.

- Here the table is divided into two parts:
  - The upper part shows the conditions and
  - The lower part shows what actions are taken.
  - Each column of the table is a rule.

Conditions				
Valid selection	No	Yes	Yes	Yes
New member	-	Yes	No	No
Renewal	-	No	Yes	No
Cancellation	-	No	No	Yes
Actions				
Display error message	x	-	-	-
Ask member's details	-	x	-	-
Build customer record	-	x	-	-
Generate bill	-	x	x	-
Ask member's name & membership number	-	-	x	x
Update expiry date	-	-	x	-
Print cheque	-	-	-	x
Delete record	-	-	-	x

— From the above table you can easily understand that, if the valid selection condition is false then the action taken for this condition is 'display error message'.
— Similarly, the actions taken for other conditions can be inferred from the table.

UNIT-4- Software Design

Basic Concepts in 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:** High-level design means identification of different modules and the control relationships, interface among them.
  - **Detailed 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.

Difference between analysis and design

- The aim of **analysis** is to understand the problem with a view to eliminate any deficiencies in the requirement specification such as incompleteness, inconsistencies, etc. The model which we are trying to build may be or may not be ready.
- The aim of **design** is to produce a model that will provide a seamless transition to the coding phase, i.e. once the requirements are analyzed and found to be satisfactory, a design model is created which can be easily implemented.

Characteristics of a good software design

- The characteristics are listed below:
  - **Correctness:** The software which we are making should meet all the specifications stated by the customer.
  - **Usability/ Learnability:** The amount of efforts or time required to learn how to use the software should be less. This makes the software user-friendly even for IT-illiterate people.
  - **Reliability:** The software product should not have any defects. Not only this, it shouldn't fail while execution.
  - **Efficiency:** This characteristic relates to the way software uses the available resources. The software should make effective use of the storage space and execute command as per desired timing requirements.
  - **Security:** With the increase in security threats nowadays, this factor is gaining importance. The software shouldn't have ill effects on data / hardware. Proper measures should be taken to keep data secure from external threats.
  - **Safety:** The software should not be hazardous to the environment/life.
  - **Understandability:** A good design is easily understandable.
    - determines goodness of design:
    - a design that is easy to understand:
      - also easy to maintain and change.
    - Unless a design is easy to understand,
      - tremendous effort needed to maintain it
      - We already know that about 60% effort is spent in maintenance.
  - **Maintainability:** It should be easily open to change.
- Use consistent and meaningful names for various design components,
- Design solution should consist of a cleanly decomposed set of modules **(modularity).**
- Different modules should be **neatly arranged in a hierarchy** in a neat tree-like diagram.

Modularity

- Modularity is a fundamental attributes of any good design.
  - Decomposition of a problem cleanly into modules (by divide and conquer principle.)
  - Modules are almost independent of each other
- If modules are independent:

- modules can be understood separately,
- so it can reduces the complexity greatly.

- In technical terms, modules should display:
  - **high cohesion and low coupling.**
  - **Neat arrangement** of modules in a hierarchy means:
    - **low fan-out**
    - **abstraction**

Cohesion and Coupling:

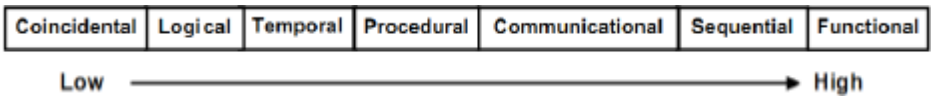
- Cohesion is a measure of:
  - Functional strength of a module.
  - A cohesive module performs a single task or function.
- Coupling between two modules:
  - A measure of the degree of interdependence or interaction between the two modules.

Cohesion:

- The primary characteristics of neat module decomposition are **high cohesion** and **low coupling.**
- As we know it is a measure of functional strength of a module.
- A module having high cohesion and low coupling is said to be functionally independent of other modules.
- A functionally independent module has minimal interaction with other modules.

Classification of cohesion

The different classes of cohesion that a module may possess are as follows:



Coincidental cohesion:

- A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely. i.e. the module contains a random collection of functions
- For example, different functions like get-input, print-error, and summarize-members functions are grouped into one module. The grouping does not have any relevance to the structure of the problem.

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:

- A module is said to exhibit 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.
- Ex: The set of functions responsible for initialization, start-up, shutdown of some process, etc. exhibit temporal cohesion.

Procedural cohesion:

- A module is said to possess procedural cohesion, if the set of functions of the module in which certain sequence of steps have to be carried out for achieving an objective, e.g. the algorithm for decoding a message.

Communicational cohesion:

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

Sequential cohesion:

- A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next.

Functional cohesion:

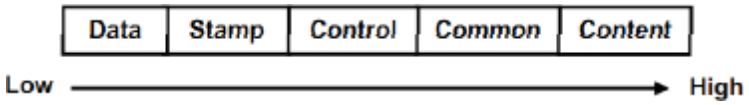
- A module is said to functional cohesion, if different elements of a module cooperate to achieve a single function.
- For example, a module containing all the functions required to manage employees' pay-roll.

Coupling

- Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules.
- A module having high cohesion and low coupling is said to be functionally independent of other modules. If two modules interchange large amounts of data, then they are highly interdependent.

Classification of Coupling

The classification of the different types of coupling will help to quantitatively estimate the degree of coupling between two modules. Five types of coupling can occur between any two modules. This is shown below



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, e.g. an integer, a float, a character, etc.

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. Example: a flag set in one module and tested in another module.

Common coupling:

- Two modules are common coupled, if they share data through some global data items.

Content coupling:

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

Function-oriented design

The following are the salient features of a typical function-oriented design approach:

- A system is viewed as a set of functions. Starting at this high-level view of the system, each function is successively refined into more detailed functions.
- For example, consider a function create-new-library-member which may consist of the following sub-functions:
  - assign-membership-number
  - create-member-record
  - print-bill
- Each of these sub-functions may be split into more detailed sub functions and so on.
- The system state is centralized and shared among different functions.

Object-oriented design

- In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities).
- The state is decentralized among the objects and each object manages its own state information.
  - For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data. In fact, the functions defined for one object cannot refer or change data of other objects.

FUNCTIONAL-ORIENTED VERSUS THE OBJECT-ORIENTED APPROACH

S. No	Functional-oriented Approach	Object-oriented Approach
1.	In the functional-oriented design approach, the basic abstractions, which are given to the user, are real world functions, such as sort, merge,track, display, etc.	In the object-oriented design approach, the basic abstractions are not the real-world functions, but are the data abstraction where the real-world entities are represented, such as picture, machine, radar system, customer, student, employee,etc.
2.	In function-oriented design, functions are grouped together by which a higher-level function is obtained. An example of this technique is SA/SD.	In this design, the functions are grouped together on the basis of the data they operate on, such as in class person, function displays are made member functions to operate on its data members such as the person name, age,etc.
3	In this approach, the state information is often represented in a centralized shared memory.	In this approach, the state information is not represented in a centralized shared memory but is implemented/distributed among the objectsof the system.

Context diagram:

- The context diagram is the most abstract data flow representation of a system.
- It represents the entire system as a single bubble.
- This bubble is labelled according to the main function of the system.
- The various external entities with which the system interacts and the data flow occurring between the system and the external entities are also represented.
- The data input to the system and the data output from the system are represented as incoming and outgoing arrows.
- These data flow arrows should be annotated with the corresponding data names.
- The context diagram is also called as the level 0 DFD.
- To develop the context diagram of the system, it is required to analyze the SRS document.

DFD(Data Flow diagram or Bubble Chart) model of a system:

- A DFD model of a system is graphical transformation of the data input to the system to the final result through a hierarchy of levels.
- A DFD starts with the most abstract definition of the system (lowest level) and ends with more detailed (higher level DFD).
- To develop a higher-level DFD model, processes are decomposed into their sub-processes and the data flow among these sub-processes is identified.
- To develop the data flow model of a system, first the most abstract representation of the problem is to be worked out.
- The most abstract representation of the problem is also called the context diagram. After, developing the context diagram, the higher-level DFDs have to be developed.

Symbols Used for Constructing DFDs

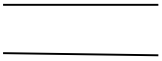
There are different types of symbols used to construct DFDs. The meaning of each symbol is explained below:

1. ***Function symbol.*** A function is represented using a circle. This symbol is called a process or a bubble and performs some processing of input data.

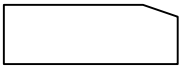
2. ***External entity.*** A square defines a source or destination of system data. External entities represent any entity that supplies or receives information from the system but is not a part of the system.

3. ***Data-flow symbol.*** A directed arc or arrow is used as a data-flow symbol. A data-flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow.

4. ***Data-store symbol.*** A data-store symbol is represented using two parallel lines. A logical file can represent either a data-store symbol, which can represent either a data structure, or a physical file on disk. Each data store is connected to a process by means of a data-flow symbol. The direction of the data-flow arrow shows whether data is being read from or written into a data store.



5. ***Output Symbol.*** It is used to represent data acquisition and production during human-computer interaction.



It's very useful during software requirements analysis. The DFDs have basic two levels of development, they are as follows –

1) A Level 0 DFD, also known as Fundamental System Model or Context Model, represents the entire system as a single bubble with incoming arrowed lines as the input data and outgoing arrowed lines as output.

2) A Level 1 DFD might contain 5 or 6 bubbles with interconnecting arrows. Each process represented here is the detailed view of the functions shown in the level 0 DFD.
   - Here the rectangular boxes are used to represent the external entities that may act as input or output outside the system.
   - Round Circles are used to represent any kind of transformation process inside the system.
   - Arrow headed lines are used to represent the data object and its direction of flow.

Following are some guidelines for developing a DFD:-

1) Level 0 DFD should depict the entire software system as a single bubble.
2) Primary input and output should be carefully noted.
3) For the next level of DFD the candidate processes, data objects and stores should be recognized distinctively.
4) All the arrows and bubbles should be labeled with meaningful names.
5) Information flow continuity should be maintained in all the levels.
6) One bubble should be refined at a time.

Data dictionary for a DFD model:

- Every DFD model of a system must be accompanied by a data dictionary.
- It lists all data items appearing in the DFD model of a system.
- The data items listed include all data flows and the contents of all data stores appearing on the DFDs in the DFD model of a system.

Advantages of Data Dictionary

- It is a mechanism for name management – Many people may have to invent names for entities and relationships when developing a large system model. These names should be used consistently and should not clash. The data dictionary software can check for name uniqueness where necessary and warn requirements analysts of name duplications.
- It serves as a store of organizational information – As the system is developed information that can link analysis, design, implementation and evolution is added to the data dictionary, so that all information about an entity is in one place.

Advantages of data flow diagram:

- A simple graphical technique which is easy to understand.
- It helps in defining the boundaries of the system.
- It is useful for communicating current system knowledge to the users.
- It is used as the part of system documentation file.
- It explains the logic behind the data flow within the system.

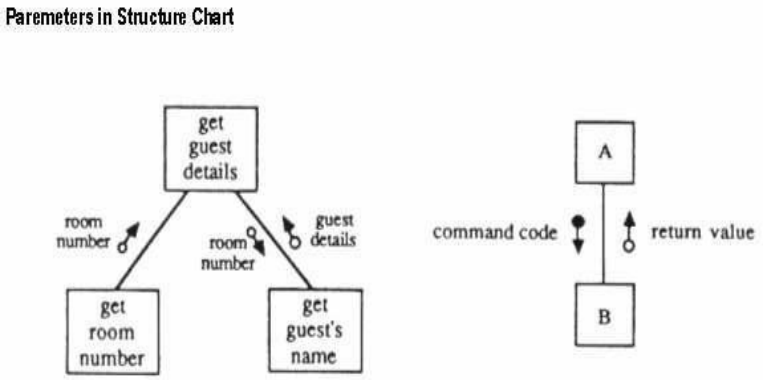
Shortcomings of the DFD model:

It suffers from several shortcomings. The important shortcomings of the DFD models are the following:

- DFDs said to be imprecise, because the function performed by a bubble is judged from its label. But, a short label may not capture the entire functionality of a bubble
- Control aspects are not defined by a DFD.
- A DFD model does not specify the order in which the different bubbles are executed.
- The method of carrying out decomposition to arrive at the successive levels depends on the choice and judgment of the analyst.
    - Due to this reason, even for the same problem, several alternative DFD representations are possible.
- The data flow diagramming technique does not provide any specific guidance as to how exactly to decompose a given function into its sub-functions.

Structure Charts

- They are used during architectural design for documenting hierarchical structure, parameters and interconnections in a system.
- In a structure chart a module isrepresented by a box with the module name written in the box.
- An arrow from a module A to a moduleB represents that the module A invokes module B. the arrow is labeled by the parameters received by B as input and the parameters returned by B.



Unit-5- Understanding the principle of User Interface Design

5.1 Rules for UID
5.2 Interface design models
5.3 UID Process and models
5.4 Interface design activities defining interface objects, actions and the design issues
5.5 Compare the various types of interface
5.6 Main aspects of Graphical UI, Text based interface

User Interface Design:

- Creates effective communication medium between a human and a computing machine
- Provides easy and spontaneous access to information as well as efficient interaction and control of software functionality

Golden Rules User Interface Design:

- Place the user in control
- Reduce the user's memory load
- Make the interface consistent

Place the User in Control

Define interaction modes in a way that does not force a user into unnecessary or undesired actions.

- Provide for flexible interaction.
- Allow user interaction to be interruptible and undoable.
- Streamline interaction as skill levels advance and allow the interaction to be customized.
- Hide technical internals from the casual user.
- Design for direct interaction with objects that appear on the screen.

Reduce the User's Memory Load

- Reduce demand on short-term memory.
- Establish meaningful defaults.
- Define shortcuts that are intuitive.
- The visual layout of the interface should be based on a real
- World metaphor.
- Disclose information in a progressive fashion.

Make the Interface Consistent

- Allow the user to put the current task into a meaningful context.
- Maintain consistency across a family of applications.
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

User Interface Design Models

- **User model —** a profile of all end users of the system.
- **Design model —** a design realization of the user model.
- **Mental model (system perception) —** the user's mental image of what the interface is.
- **Implementation model —** the interface "look and feel" coupled with supporting information that describe interface syntax and semantics.

Interface Design Steps:

- Using information developed during interface analysis, define interface objects and actions (operations).
- Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
- Depict each interface state as it will actually look to the end-user.

- Indicate how the user interprets the state of the system from information provided through the interface.

Design Issues:

- Response time
- Help facilities
- Error handling
- Menu and command labeling
- Application accessibility
- Internationalization

Characteristics of a user interface

- It is very important to identify the characteristics 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.

Speed of use:

- It is determined by the time and user effort necessary to initiate and execute different commands.
  - It indicates how fast the users can perform their intended tasks. The time and user effort necessary to initiate and execute different commands should be minimal.
- This characteristic of the interface is referred to as productivity support of the interface.

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.

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.

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.

Support for multiple skill levels:

- A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users.

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.

User guidance and on-line help:

- Users seek guidance and on-line help when they either forget a command or are unaware of some features of the software.
- Whenever users need guidance or seek help from the system, they should be provided with the appropriate guidance and help.

Types of user interfaces

User interfaces can be classified into the following three categories:

- Command language based interfaces
- Menu-based interfaces
- Direct manipulation interfaces

Command Language-based Interface

- It 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 them in appropriately whenever required.
  - A simple command language-based interface might simply assign unique names to the different commands.
  - However, a more sophisticated command language-based interface may allow users to compose complex commands by using a set of primitive commands.
- Such a facility to compose commands dramatically reduces the number of command names one would have to remember.
- Thus, a command language-based interface can be made concise requiring minimal typing by the user. Command language-based interfaces allow fast interaction with the computer.

Menu-based Interface

- It does not require the users to remember the exact syntax of the commands.
- It is based on recognition of the command names, rather than recollection.
  - Further, in this the typing effort is minimal as most interactions are carried out through menu selections using a pointing device.
- This factor is an important consideration for the occasional user who cannot type fast.

Direct Manipulation Interfaces

- It presents the interface to the user in the form of visual models (i.e. icons or objects). For this reason, it sometimes called as iconic interface.
- In this type of interface, the user issues commands by performing actions on the visual representations of the objects,
  - E.g. pull an icon representing a file into an icon representing a trash box, for deleting the file. Important advantages of iconic interfaces include the fact that the icons can be recognized by the users very easily, and that icons are language-independent.
- However, direct manipulation interfaces can be considered slow for experienced users. Also, it is difficult to give complex commands using a direct Manipulation interface.

Graphical User Interface vs. Text-based User Interface

The following comparisons are based on various characteristics of a GUI with those of a text-based user interface.

- In a GUI multiple windows with different information can simultaneously be displayed on the user screen. This is perhaps one of the biggest advantages of GUI over text-based interfaces since the user has the flexibility to simultaneously interact with several related items at any time and can have access to different system information displayed in different windows.
- 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 be deleting is intuitively very appealing and the user can instantly remember it.
- A GUI usually supports command selection using an attractive and user-friendly menu selection system.
- 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
- Whereas a Text based interface can be implemented even on a cheap alphanumeric display terminal.

UNIT-6: Understanding The Principle Of S/W coding-2020

CODING

- Coding phase is starts once the design phase is complete.
  - Input to the coding phase is the design document.
  - At the end of the design phase, we obtain
    - Module Structure (Structure Chart)
    - Module Specification (DS & Algorithm of each module is specified).
- To develop a good s/w, it needs to follow some standard. This is known as Coding standard.
- Coding standard are required and the developers strictly follow these standards, because
  • It gives a **uniform appearance to the codes** written by different engineers.
  • It **enhances code understanding**.
  • It encourages **good programming practices.**
- Beside the coding standards the s/w companies also suggest several coding guidelines.

Coding Standards and Guidelines

- Good software development organizations usually develop their own coding standards and guidelines.

Coding Standards:

The following are some representative coding standards

1. **Rules for limiting the use of global variable:** These rules list what types of data can be declared global and what cannot.
2. **Contents of the headers preceding codes for different modules:** Standard for header data should have Name of the module, Date of creation, Author's name, Modification history, etc.
3. **Naming conventions for global variables, local variables, and constant identifiers:** for Example: global variable names always start with a capital letter, local variable names are made of small letters, and constant names are always capital letters.
4. **Error return conventions and exception handling mechanisms:** This shows how the error conditions are reported. For example, an error condition should either return a 0 or 1 consistently.

Coding Guidelines:

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

1. Do not use a coding style that is too clever or too difficult to understand
2. Avoid obscure (Unclear) side effects
3. Do not use an identifier for multiple purposes.
4. The code should be well-documented.

5. The length of any function should not exceed 10 source lines.
6. Do not use GOTO statements.

Code Review

- Code review for a model is carried out after the module is successfully compiled and the all the syntax errors have been eliminated.
- Code reviews are extremely cost-effective technique for reduction in coding errors and to produce high quality code.
- Normally, two types of reviews are carried out on the code of a module.
    - Code Inspection and
    - Code walk through.

Code Walk Through:

- It is an informal code analysis technique.
- A few members of the development team are given the code few days before the walk through meeting to read and understand code.
- Each member selects some test cases and simulates execution of the code by hand (i.e. trace execution through each statement and function execution).
- The main objectives of the walk through are **to discover the algorithmic and logical errors in the code.**

Code Inspection:

- In contrast to code walk through, the aim of code inspection is **to discover some common types of errors caused due to oversight and improper programming**.
- In other words, during code inspection the code is examined or simulation of code execution done by the system.
- Following is a list of some classical programming errors which can be checked during code inspection:
  • Use of uninitialized variables.
  • Jumps into loops.
  • Non-terminating loops.
  • Array indices out of bounds, etc.

Software documentation

Software documents are a vital part of good software development. Good documents are very useful for the following purposes:

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

The various kinds of documents are users' manual, software requirements specification (SRS) documents, design documents, test documents, installation manual, etc have to develop.
Different types of software documents can broadly be classified into the following:

- **Internal documentation:** It is the source code itself.
- **External documentation:** It is provided through various types of supporting, documents such as users' manual, software requirements specification document, design document, test documents, etc.

Testing:

- It is intended process of finding errors.
- Program is tested with a set of test inputs (or test cases) and observing if the program behaves as expected or not. If the program fails to behave as expected, then it requires debugging or correction.
- Some commonly used terms associated with testing are:
    - **Failure:** This is a manifestation of an error (or defect or bug). But, the presence of an error may not necessarily lead to a failure.
    - **Test case:** This is the triplet [I, S, O], where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system.

• **Test suite:** This is the set of all test cases with which a given software product is to be tested.

Aim of testing

- The aim of the testing process is to identify all defects existing in a software product.
- Thus testing provides a practical way of reducing defects in a system and increasing the users' confidence in a developed system.

Design of test cases:

- The test suite should be carefully designed rather than randomly. Because random selection does not guarantee that all of the errors in the system will be detected.
- Therefore, systematic approaches should be followed to design an optimal test suite. In an optimal test suite, each test case is designed to detect different errors.
- There are essentially two main approaches to design test cases:
  - **Black-box testing approach(Functional testing)**
  - **White-box testing (or glass-box) approach(structural testing)**

Black box testing

- In this approach, test cases are designed using only the functional specification of the software.
- Knowledge of the internal structure of the software is not required.
- For this reason, this testing is known as functional testing.
- In it, test cases are designed to test the input/output values only and no knowledge of design or code is required.
- The following are the two main approaches to designing black box test cases.
  • Equivalence class partition
  • Boundary value analysis

Equivalence Class Partitioning:

- In this approach, the domain or range of input values to a program is partitioned into a set of equivalence classes.
- Partitioning is done in such a way that it should produce then one valid and two invalid equivalence classes.

Example: For a software that computes the square root of an input integer which can assume values in the range of 0 to 5000.
There are three equivalence classes:
1. The set of negative integers i.e. -∞ to -1.
2. The set of integers in the range of 0 and 5000, and
3. The integers larger than 5000.

Therefore, the test cases must include representatives for each of the three equivalence classes and a possible test set can be: {-10, 50, 6000}.

Boundary Value Analysis

- It is seen from the practice that error frequently occurs at the boundaries.
- The reason behind such errors might purely be due to psychological factors.
- For example, programmers may improperly use < instead of <=, or conversely <= for <. Boundary value analysis leads to selection of test cases at the boundaries of the different equivalence classes.

Example: For a function that computes the square root of integer values in the range of 0 and 5000, the test cases must include the following values: {0, - 1, 5000, 5001}.

White box testing

- It is used to test the program code, code structure and the internal design flow.
- The internal logics, such as control structures, control flow, and data structures are considered during the white-box testing.

Different white box strategies are:

Statement coverage

- This strategy aims to design test cases so that every statement in a program is executed at least once. The idea behind it is that unless a statement is executed, it is very hard to determine whether it is correct or wrong.

Example:

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

The test cases/set for x, y: {(x=3, y=3), (x=4, y=3), (x=3, y=4)}, we can exercise the program such that all statements are executed at least once.

Branch coverage

- In this strategy, test cases are designed to test each branch condition and each branch would returns true and false values.
- It is also known as edge testing.
- It is a stronger testing strategy as compared to the statement coverage-based testing.

For above computation algorithm,
The test cases can be {(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)}.

Condition coverage

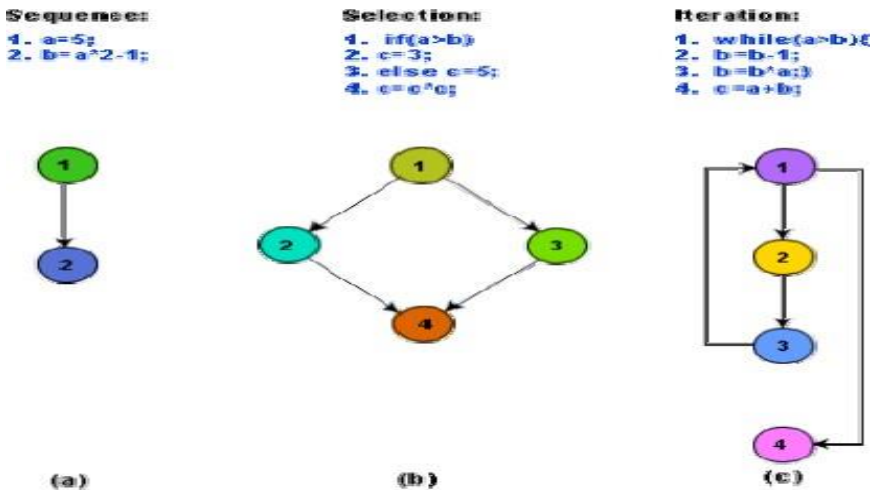
- This testing done on composite conditional expression. Test cases are designed in such a way that each conditional expression should return both true and false values.
- For example, in the conditional expression ((c1.and.c2).or.c3), the components c1, c2 and c3 are each made to assume both true and false values.

Path coverage

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

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.



Mutation testing

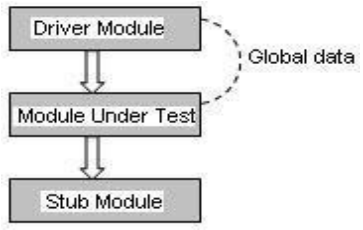
- In mutation testing, the software is first tested by using an initial test suite. After the initial testing is complete, mutation testing is taken up.
- It is a fault based technique or fault simulation technique.
- Multiple copies of a program are made and each copy is altered. This altered copy is called a mutant. Mutants are executed with test data to determine whether the test data are capable of detecting the change between the original program and the mutated program.
- A mutant that is detected by a test case is termed "killed" and the goal of mutation procedure is to find a set of test cases that are able to kill groups of mutant programs.
- IF (test suit kills a mutant) THEN test suit can kill all similar mutants.
- IF (mutant is alive) THEN test suit could not catch similar errors.
- Result is: **updating the test suite.**

Levels of Testing:

- There are 3 levels of testing:
  - Unit Testing
  - Integration Testing
  - System Testing

Unit testing:

- Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation/individual.
- In order to test a single module, a complete environment is needed to provide all that is necessary for execution of the module.



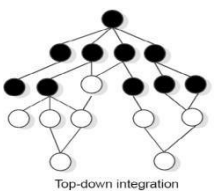
- Modules required the necessary environment is usually not available until they too have been unit tested;
  - Stubs and drivers are designed to provide the complete environment for a module.
- The role of stub and driver modules is pictorially shown in figure.
- A stub procedure is a dummy procedure that has the same I/O parameters as the given procedure.
- A driver module contains the nonlocal data structures accessed by the module under test.

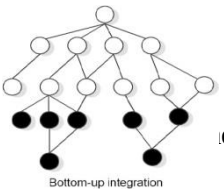
Integration Testing

- The purpose of unit testing is to determine whether each independent module is correctly tested or not.
  - This gives little chance to determine that the interface between modules is also correct, and for this reason integration testing must be performed.
- The primary objective of integration testing is to test the module interface in order to ensure that there are no errors in the parameter passing, when one module invokes another module.

Different approaches for integration testing:

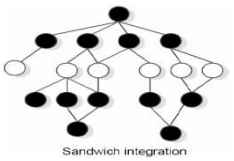
- Big-bang approach
- Top-down approach
- Bottom–up approach
- Mixed approach
- **Big bang approach** is the simplest integration testing approach:
  –All the modules are simply put together and tested.
  – This technique is used only for very small systems.
- **In top-down approach:**
  – Testing waits till all top-level modules are coded and unit tested.
  – It starts with the main routine:
    - After the top-level 'skeleton' has been tested:





- **In bottom-up approach:**
  - Testing can start only after bottom level modules are ready.

- **Mixed (or Sandwiched) integration testing:**
  - Uses both top-down and bottom-up testing approaches.
  - Most common approach.



Phased versus Incremental Integration Testing:

- The diff. Integration testing strategies we have discussed. It can either be of phased or be of incremental integration type.
- In incremental integration testing:
  - Only one new module is added to the partial system each time.
- In phased integration:
  - A group of related modules are added to the partially integrated system each time.
- Phased integration requires less number of integration steps as compared to the incremental integration approach.
- However, when failures are detected,
  - it is easier to debug if using incremental testing
    - Since errors are very likely to be in the newly integrated module.

System testing:

- System tests are designed to validate a fully developed system to assure that it meets its requirements.
- There are essentially three main kinds of system testing:
- **Alpha Testing:**
  - Alpha testing refers to the system testing carried out by the test team within the developing organization.
  - It **is conducted at the developer's** site by end users.
- **Beta testing:**
  - Beta testing is the system testing performed by a select group of friendly customers.
  - It **is conducted at the end-user's site** and done by the end user.
  - It is a "**live" application of the software in** an environment that cannot be controlled by the developer.
- **Acceptance Testing.**
  - This testing performed by the customer himself:
    - To determine whether the system should be accepted or rejected.

Need for debugging:

- Debugging is defined as a process of analyzing and removing the error.
- Identifying errors in a program code and then fix them are known as debugging.

Debugging approaches:

The following are some of the approaches popularly adopted by programmers for debugging.

Brute Force Method:

- This is the most common method of debugging but is the least efficient method.
- In this approach, the program is loaded with **print statements** to print the intermediate values with the hope that some of the printed values will help to identify the statement in error.
- This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger).

Backtracking:

- This is also a fairly common approach.
- In this approach, beginning from the statement at which an error symptom has been 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.

Program Slicing:

- This technique is similar to back tracking. Here the search space is reduced by defining slices.
- A slice of a program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.

Debugging guidelines:

It is often carried out by programmers based on their ingenuity. The following are some general guidelines for effective debugging:

- Debugging usually requires a thorough understanding of the program design.
- Debugging may sometimes require full redesign of the system.
  - A common mistake novice (new) programmers often make:
    - Is that they do not attempt to fixing the error but only its symptoms.
- Be aware of the possibility that an error correction may introduce new errors.
- After every round of error-fixing, regression testing must be carried out.

Performance Testing

- Performance testing is a non-functional testing technique.
- Performance testing is the process of determining the speed or effectiveness of software program or system.
- It is done to determine the system parameters or quality attributes of the system such as response, stability, scalability, reliability and resource usage under various workloads.

Error Seeding

- It is a process, by which we can intentionally or consciously adding of errors to the source code.
- After that, test runs are done to detect errors.
- To evaluate the total number of errors detected, we have to calculate the ratio between actual and artificial errors.
- i.e. it is used to evaluate the amount of residual errors after adding the errors to the source code.

Some General Issues Associated with Testing

- In this section, we will discuss two general issues associated with testing.
- They are:
  - Test Documentation
  - Regression Testing

Test Documentation:

- It is generated towards the end of testing is the test summary report.
- It will specify:
  - How many tests have been applied to a subsystem,
  - How many tests have been successful,
  - How many tests have been unsuccessful?

Regression testing:

- This testing is done when some modification is made in the software, it is due to either enhancement or to remove some bugs.
- In Regression testing, the system is run against the old test suite to ensure that no new bug has been introduced due to the change or the bug fix.
- It is performed in the maintenance or development phase.

UNIT-7- Understanding the importance of software Reliability

7.1 Importance of S/W reliability
7.2 Distinguish between the different reliability Metrics
7.3 Reliability growth modelling
7.4 Characteristics of quality software
7.5 Evolution of s/w quality management system

Software reliability

- Reliability of a software product shows its **trustworthiness or dependability.**
- It is **observer-dependent.**
- It can also be defined as the probability of **the product working "correctly" over a given period of time.**
- If a software product having a large number of defects, then we call it as **unreliable**.
- It is also clear that the reliability of a system improves, if the number of defects in it is reduced.
- Reliability of a product depends not only on the number of errors but also on the exact location of the errors.

Reasons for software reliability being difficult to measure

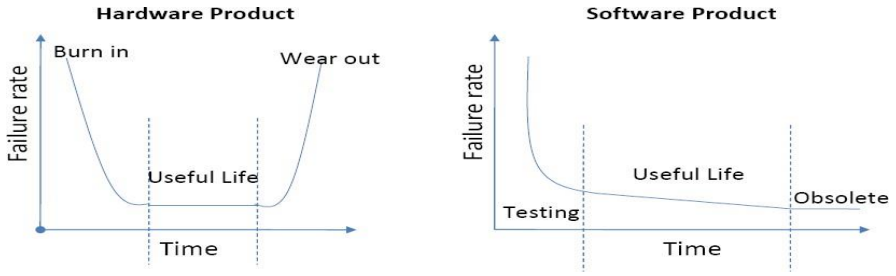
The reasons why software reliability is difficult to measure can be summarized as follows:

- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The reliability of a software product is highly observer-dependent.
- The reliability of a product keeps changing as errors are detected and fixed.
- Software reliability depends on its trustworthiness or dependability.
- Reliability improves implies defects reduces.

Hardware reliability vs. software reliability:

Reliability behaviour for hardware and software are very different. For example,

- Most hardware failures are due to component wear and tear. To fix hardware faults, one has to either replace or repair the failed part.
- On the other hand, a software product would continue to fail until or unless the error is tracked and fixed.
- For this reason, when hardware is repaired its reliability is maintained; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug introduces new errors).
- We can say differently that, hardware reliability is concerned with stability. On the other hand, software reliability is concerned with reliability growth.
- The change of failure rate over the product lifetime for a typical hardware and a software product are given in figure.
    - For **hardware products**, it can be observed that failure rate is high initially but decreases as the faulty components are identified and removed. The system then enters its useful life. After some time the components wear out, and the failure rate increases. The hardware reliability characteristicover time is "bath tub" shape.
    - For **software** the failure rate is at its highest during integration test. As the system is tested over the time, more and more errors are identified and removed and then failure rate reduces. As the software becomes obsolete (out of date) no error corrections occurs and the failure rate remains unchanged.



Reliability metrics

The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the SRS (software requirements specification) document. To do this, some metrics are needed to express the reliability of a software product. There are six reliability metrics which can be used to quantify the reliability of software products.

- **Rate of occurrence of failure (ROCOF):**
  - It measures the frequency of occurrence of unexpected behaviour (i.e. failures). It measure by observing the behaviour of a software product in operation over a specified time interval and then recording the total number of failures occurring during the interval.
- **Mean Time To Failure (MTTF):**
  - It is the average time between two successive failures, observed over a large number of failures.
- **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, if MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours.
- **Probability of Failure on Demand (POFOD):**
  - POFOD measures the likelihood of the system failing when a service request is made. For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure.
- **Availability:**
  - Availability of a system is a measure of how likely shall the system is 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 consider the repair time (down time) of a system when a failure occurs.
  - This metric is important for telecommunication systems, and operating systems.

Reliability growth models

- It is a mathematical model of how software reliability improves as errors are detected and repaired. It can be used to predict or attained a particular level of reliability.
- Thus, reliability growth modelling can be used to determine when to stop testing to attain a given reliability level.
- Although several different reliability growth models have been proposed, but we will discuss only two very simple reliability growth models.
  - **Step Function Model.**
  - **Jelinski and Moranda Model.**

Step Function Model:

- The simplest reliability growth model is a step function model.
- It is assumed that the reliability increases by a constant increment each time an error is detected and repaired.
- This model assumes that all errors contribute equally to reliability growth. But this is highly unrealistic because we know that correction of different types of errors contributes differently to reliability growth.

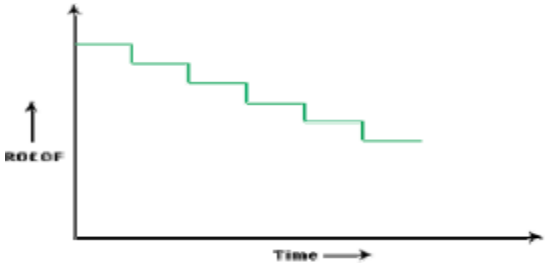


Fig.: Step function model of reliability growth

Jelinski and Moranda Model:

- This model assumes reliability does not increase by constant amount each time an error is repaired.
- It assumes reliability increases due to fixing of errors proportional to the number of errors remaining in the system that time.
- This model is more realistic for many applications but still suffers from many shortcomings. As we know the repairing the errors initially contribute high rate of reliability growth but it slow down later on.

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 in terms of satisfaction of the requirements mentioned in the SRS document.
- For software products, "fitness of purpose" is not a wholly satisfactory definition of quality. Example: consider a software product that is functionally correct. That is, it performs all functions as specified in the SRS document. But, has an almost unusable user interface.
- Therefore, the traditional concept of quality as "fitness of purpose" for software products is not wholly satisfactory.
- The modern view/ **characteristics of a quality** associates with a software product several quality factors such as the following:

    - **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 caneasily 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, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

Software quality management system

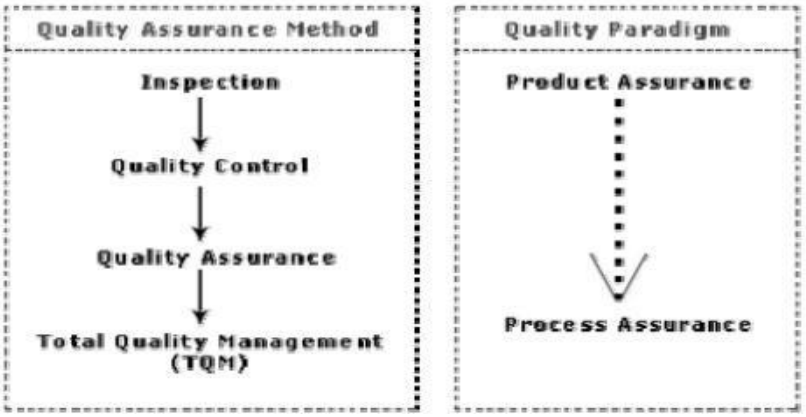
A quality management system (or quality system) is the principal methodology used by organizations to ensure that the products they develop have the desired quality. A quality system consists of the following:

- **Managerial Structure and Individual Responsibilities:**
  Every quality conscious organization has an independent quality department:
    - Performs several quality system activities.
    - Needs support of top management.
    - Without support at a high level in a company, many employees may not take the quality system seriously.
- **Quality System Activities:**
  The quality system activities encompass the following:
    - auditing of projects
    - review of the quality system
    - Development of standards, procedures, and guidelines, etc.
    - Production of reports for the top management summarizing the effectiveness of the quality system in the organization.

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 thefig.



- The initial product inspection method gave way to quality control (QC).
- **Quality control** focuses not only on detecting the defective products and eliminating them but also on determining the causes behind the defects. Thus, quality control aims at correcting the causes of errors and not just rejecting the products.
- The next in quality systems was the development of **quality assurance** principles. The basic of modern quality assurance is that if an organization's processes are good and are followed rigorously, then the products are bound to be of good quality.
    - The modern quality paradigm includes guidance for recognizing, defining, analyzing, and improving the production process.
- **Total quality management (TQM)** advocates that the process followed by an organization must be continuously improved through process measurement.
    - It goes a step further than quality assurance and aims at continuous process improvement. TQM goes beyond documenting processes to optimizing them through redesign.
- From the above discussion it can be stated that over the years the quality paradigm has shifted from product assurance to process assurance as shown in figure.

QUESTION BANK

CHAPTER-1

2MARKS

1. What is SE?
2. Define feasibility study in a software project development.
3. Explain why the spiral life cycle model is considered to be a Meta model.
4. Define software.
5. What is SDLC?
6. What is phase containment of error?
7. What is maintenance?
8. Mention the different types of software maintenance.
9. Diff. between verification and validation.

5&10 MARKS

1. Explain features of spiral model with suitable diagram.
2. Explain the different phases of classical waterfall model with suitable diagram.
3. What do you mean by software model (SDLC)? Explain iterative waterfall model with suitable diagram.
4. Explain how software is different from a program.
5. Define software maintenance. Explain the software maintenance process model.
6. What is prototype model? Explain this model with suitable diagram.

CHAPTER-2

2MARKS

1. What is risk?
2. What is COCOMO model?
3. Why COCOMO model is used?
4. Why Gantt Chart, PERT Chart are used?
5. Name two popular empirical estimation techniques.

5&10 MARKS

1. Discuss about various project size estimation metrics or explain the FP and LOC metrics.
2. What is a risk? Describe various activities associated with risk management in software development.
   OR
   Explain risk management. What are the important types of risks that a project might suffer from? Identify the best risk reduction technique.
3. Define COCOMO. Explain the basic features of COCOMO model along with the estimates.
4. Define COCOMO. Explain the features of different COCOMO model along with the estimates.
5. Explain the organization and team structure of project mgmt.
6. What are the essential activities of project planning? Explain.
7. What is the different configuration mgmt. Activities? Explain.

CHAPTER 3:-

2MARKS:-

1. What is a prototype?
2. What do you mean by SRS?

5&10 MARKS

1. What is SRS? What are the different categories of user of SRS? Explain the features of a good SRS document and its contents.
2. What are the characteristics of good SRS document?

CHAPTER 4

2MARKS

1. Write the different approaches to software design.
2. What do you mean by cohesion and coupling?
3. What do you mean by fan-in and fan-out?
4. Write the symbols used in DFD?
5. What is modularity.

5&10 MARKS

1. What are the shortcomings of DFD model?
2. Write the characteristics of good software design.
3. Distinguish between cohesion and coupling..
4. Explain the classification of cohesion and coupling.
5. Describe the methods to transform the DFD model into structure chart.

CHAPTER 5

2MARKS

1. What do you mean by user interface?

5&10 MARKS:-

1. What is user interface? Classify or discuss different types of user interface.
2. What are the characteristics of a good user interface design?

CHAPTER 6

2MARKS

1. What is testing?
2. What do you mean by debugging?
3. What is unit testing?
4. What is integration testing?
5. What is alpha and beta testing?
6. What is mutation testing?
7. What is regression testing?
8. What is stress testing?
9. What is acceptance testing?

5&10 MARKS:-

1. Briefly explain internal and external documentation.
2. How code review is conducted? Explain.
3. What is code inspection? How it is different from code walk through?
4. Explain the different methods of black box testing techniques.
5. Explain the different methods of white box testing techniques.
6. Explain white box testing and black box testing.
7. What is testing? Discuss different levels of testing.
8. Discuss different integration testing techniques.
9. What is debugging? Discuss the different approaches used for debugging.
10. What is Cyclomatic complexity? Why it is used? Explain how Cyclomatic is computed taking one example.

CHAPTER 7:-

2MARKS:-

1. Define software reliability.

5&10MARKS:-

1. What do you mean by software quality? Describe software quality mgmt system.
2. Explain different quality factor / characteristics with a software product.
3. What do you mean by hardware and software reliability? Discuss.
4. What is reliability metrics? Discuss different types of reliability metrics.
5. What is reliability growth model? Discuss different types of reliability growth model.

References

Sl.No	Name of Authors	Title of the Book	Name of the Publisher
1	Rajib Mall	Fundamentals of Software Engineering	PHI
2	Deepak Jain	Software Engineering: Principles and Practice	Oxford university press
3	Jawadekar	Software Engineering: A Prime	TMH