PNS SCHOOL OF ENGINEERING & TECHNOLOGY

Nishamani Vihar, Marshaghai, Kendrapara

LECTURE NOTES

ON

SOFTWARE ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

5TH SEMESTER

PREPARED BY

Er.Madhusmita Ram

LECTURER IN COMPUTER SCIENCE & ENGINEERING

UNIT-1

INTRODUCTION TO SOFTWARE ENGINEERING

Program:

It is the set of executable instructions intended to do specific task. Program is generally developed by single or two users.

Software:

Software is a set of instruction, data and programs used to operate computer and execute specific task. It works as interface between the user and the hardware. Software is of 2 types:

1. System Software
2. Application Software

System Software:

System software is a set of computer programs, which is designed to manage system resources. It is a collection of such files and utility programs that are responsible for running and smooth functioning of your computer system with other hardware. Ex: antivirus s/w, OS, device driver etc.

Application Software:

Application Software is a type of software that is mainly developed to perform a specific task as per the user's request. It acts as an interface between the end-user and system software.

Ex: MS office, browser, media player etc.

Program	Software
• Program is small in size. • There is minimum documentation Developer himself is the only user. • A single or may be two developer in program • Generally a program does not have user interface.	• Program is very big in size as compared to program. • There is large and extensive documentation. • There is large number of users. • Generally developed by team of developers. • Software have user interface.

Software Engineering:

It is the field of Computer Engineering that deals with the building of software system which is so large and complex that are built by a team of Engineers.

It is a discipline whose aim is the production of quality software i.e delivered on time, within the budget and that satisfied requirements.

Software Characteristics:

Software is a logical element.

Software is developed or engineered.
It is not manufactured.
Software does not wear out but it deteriorates due to change. Most software are custom built.

Software Life Cycle Models:

The goal of software engineering is to provide model and processes that need to be carried out to develop a software product.

The various phases of software life cycle are known as software development life cycle(SDLC).

The phases in SDLC are: - Prilimnary Investigation
- Software Analysis
- Software design - Software testing
- Software maintenance

Classical Waterfall Model:

This model is called linear sequential model.
The project development is divided into a sequence of well defined phases.
It is applied for long term project and well understood product requirement.
As water flows from top to bottom the phases from follows from top to bottom and there is no backtracking from one phase to previous phase.
Different phases of this model are:

1. Feasibility study
2. Requirement analysis and specification
3. Design
4. Coding and Unit testing
5. Integration and System testing
6. Maintenanance

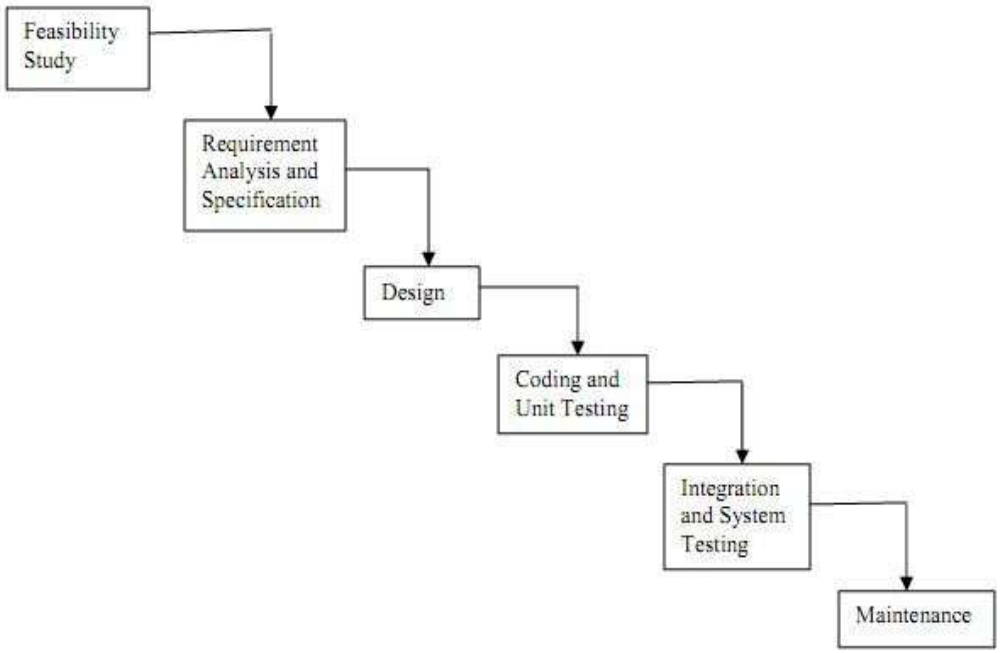


Fig 1.1 Classical Waterfall Model

1)Feasibility Study:

The main aim of feasibility study is to determine whether it would be financially, technically and operationally feasible to develop the product.

It involves the analysis of the problem and collection of all relevant information related to the product.

2)**Requirement Analysis and Specification:**

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

It consists of two activities:

i)Requirement gathering and analysis:

This activity consists of gathering the requirements and then analysing the gathered requirements.

ii)Requirement specification:

The requirements identified during the gathering and analysis activity are organised into a Software Requirement Specification(SRC) document.

3) Design:

The goal of this phase is to transform the requirements specified in the SRS document into a structure i.e suitable for implementation in some programming language.

4) Coding and Unit testing:

The purpose of this phase is to translate the software design into source code. During testing the individual unit is tested and modified.

5) Integration and System testing:

During this phase different modules are integrated in a planned manner and then tested.

Finally after all the modules are successfully integrated and tested, system testing is carried out. The goal of this testing is to ensure the requirements in the SRS document. System testing consists of 3 different types of testing

a)*α Testing:*

α Testing is the system testing performed by the development team.

b*)β Testing:*

β Testing is the system testing performed by a friendly set of customers.

c)*Acceptance Testing:*

This is the system testing performed by the customer himself. After that the customer decides whether to accept the product or to reject it.

6) Maintenance:

It includes errors correction and enhancement of capabilities. Maintenance involves following activities.

Corrective Maintenance:

This involves correcting errors that were not discovered during the product development phase.

Perfective Maintenance:

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

Adoptive Maintenance:

This is usually required for the software to work in a new environment.

Iterative Waterfall Model:

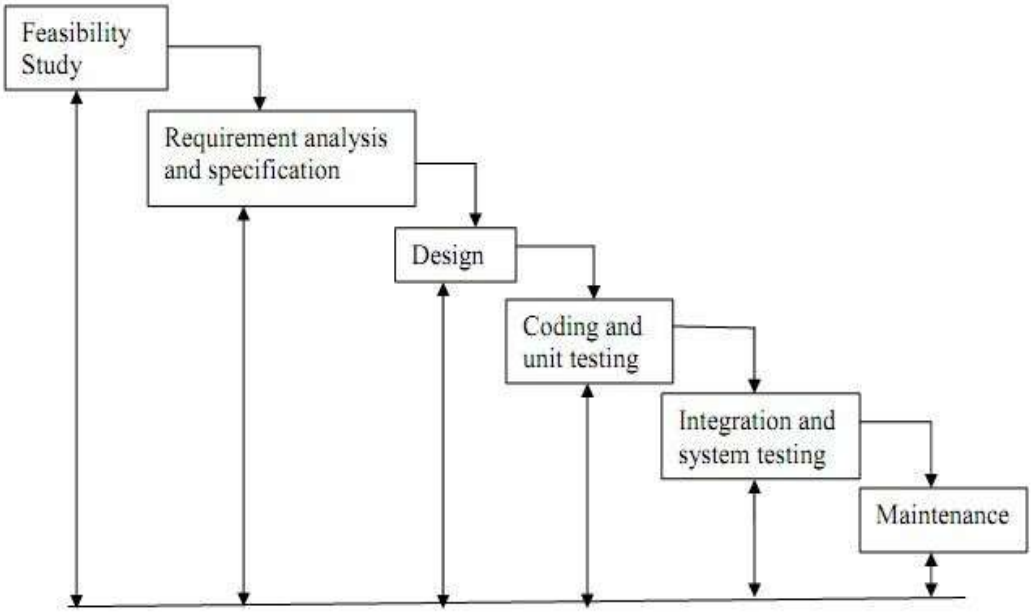


Fig. 1.2 Iterative Waterfall Mode

The classical waterfall model is theoretical.
It assumes that there is no development error in any phase but it is not practically possible.
So in Iterative waterfall model a feedback path is provided in every phase.
It allows for correction of the errors committed in a phase which are detected later.
It is better to control the errors in which phase error occurs. This is known as *phase containment error*.

Prototyping Model:

It is good for complicated and large software where there is no existing system to decide the requirements.
Prototype having limited functional capabilities, low reliability than the actual software.

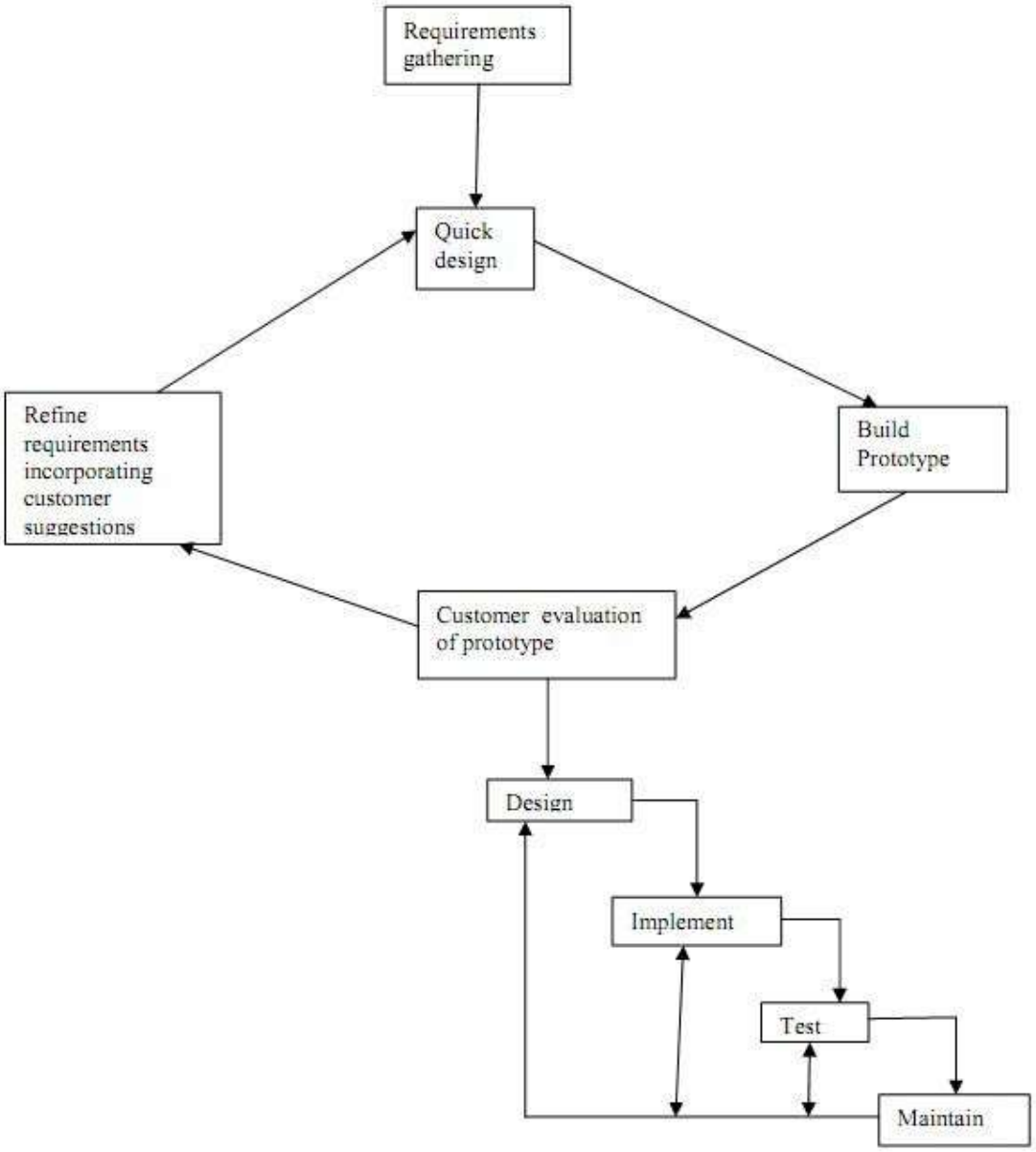


Fig. 1.3 Prototyping Model

In this model the project is built quickly and delivered to the customer for inputs and feedback.

Depending on the feedback of the customer the prototype is modified.

This cycle continues till the customer approves the prototype.

After the finalisation according to the SRS document the final product is developed using the iterative waterfall model.

This model is suited for projects where requirements are hard to determine.
This model requires extensive participation of the customers which is not always possible.

Evolutionary Model:

In this model the software is first broken into several mopdules which can be incrementally created and delivered.
The development team first develops the core module of the system.
The initial product is refined increasing level by level adding capability.

Each version of the product is fully functioning software which is more capable than the previous version.

This model is used when customer prefers to receive the product in increments rather than waiting for the full product.
It is used only for very large products.

The main disadvantage is that for most practical problems, it is difficult to divide the problems into several modules or part.

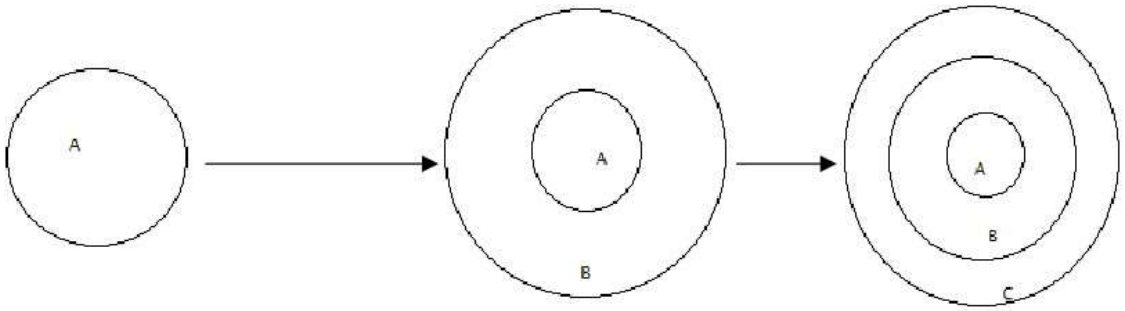


Fig.1.4 Evolutionary Model

Spiral Model:

This model of development combines the features of prototyping model, the waterfall model and other models.
The representation of this model appears like spiral with many loops.

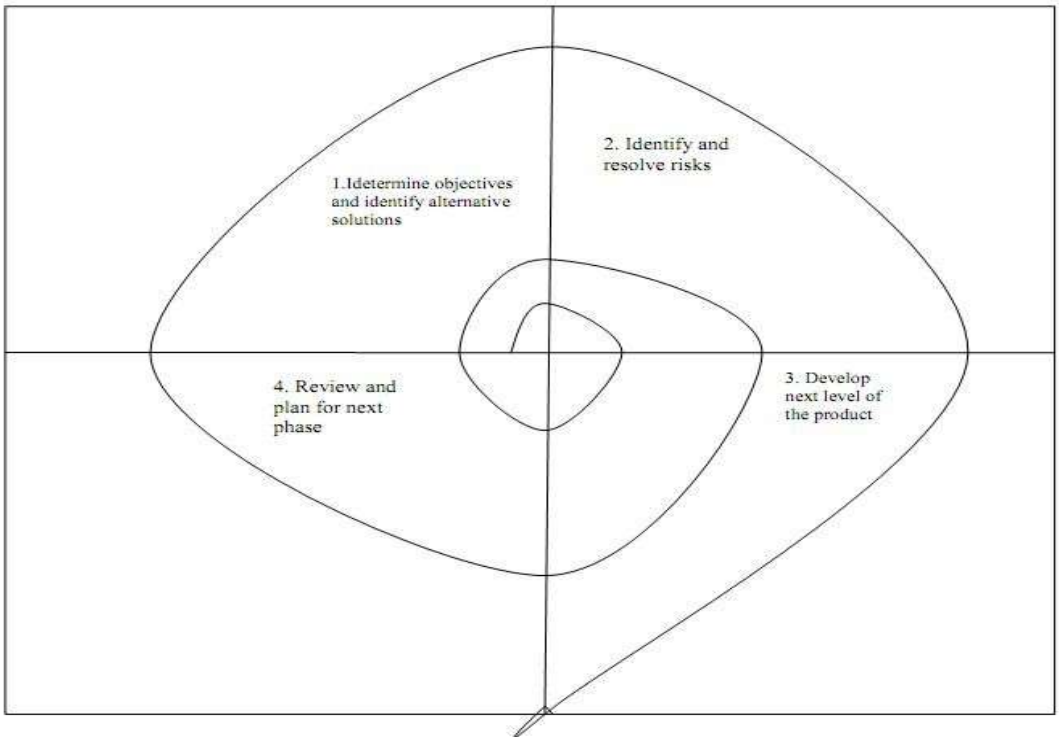


Fig. 1.5 Spiral Model

Exact number of phases through which the product is developed is not fixed.
The number of phase varies from one project to another.
Each phase is divided into 4 sectors or quadrants.
The quadrants are:

1.Planning:
Identify the objectives of the phase and the alternative solution possible for the phase and constraints. *2.Risk Analysis:*
Analyse alternatives and attempt to identify and resolve the risk involved.
3.Product Development and testing the product:
This includes customer assessment and customer evaluation.

Avantage:

- It provides early indication of risks.
- High risk functions are developed first.
- Early and frequent feedback from users.

Disadvantage:

- Model is complex.
- Risk assessment requires expertise, experience. ➢ Spiral may continue indefinitely.

Questions :

1. What is the difference between software and hardware?
2. What is the difference between program and software?
3. What is phase containment of error?
4. Why classical waterfall model is called as theoretical model?
5. What is the difference between prototyping model and evolutionary model?
6. Why spiral model is difficult to implement?

UNIT- 2

SOFTWARE PROJECT MANAGEMENT

Project Management:

The goal of the software project management is to enable a group of S/W engineers to develop a software product within budget and on schedule.

Job Responsibility of Software Project Manager:

Software managers are responsible for planning and scheduling of project development.

The managers must decide what resource are required when and how the goals to be achieved. The project manager takes responsibility for:

- Project proposal writing
- Project cost estimation
- Project staffing
- Project monitoring and control
- Risk management
- Interface with the client
- Managerial report writing and presentation

Project managers monitor the progress to check the development is on time and within budget.

Skills Necessary for Software Project Manager:

The project manager is the captain of the ship. The managers must maintain all aspect of the software. So the skills necessary are:

- Good quantitative judgement and decision making capabilities.
- Good knowledge of latest software project management techniques.
- Good communication skill and previous experience in managing similar projects.

Project Planning:

The project planning consists of the following activities:

1. Estimate the size of the project
2. Estimation of the cost and duration of the project (It depends on the size of the project).
3. Staff organisation and staffing plan.
4. Scheduling man power and other resources.
5. The amount of t computing resources.
6. Risk identification and analysis

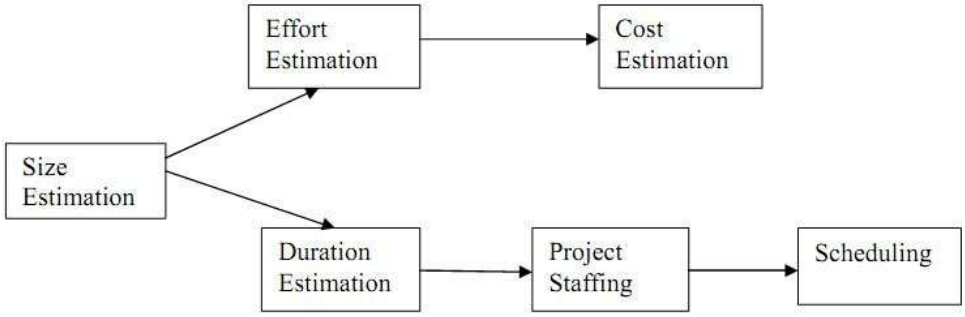


Fig. 2.1Project Planning

Project Size Estimation Metrics:

The project size is a measure of the problem complexity in terms of the effort and time required to develop the product. Two metrics are widely used to estimate the size such as

a) Lines of Code(LOC)
b) Function Point (FP)

a)Lines of Code(LOC):

LOC is defined as the no. of lines of code in software excluding the comments and blank lines. LOC depends on the programming language chosen for the project. It can't be accurate.

Disadvantage:

- LOC is `language dependent.
- LOC measures do not relate to the quality and efficiency of the code.
- It is very difficult to accurately estimate LOC in the final product from the problem specification.

b)Function Point:

Function point measures the size by a quantifying functionality provided to the user. It is independent of the computer language, development methodology, technology used. The function point analysis is designed to measure business applications(not scientific application). Function point can be estimated early in analysis and design.

Parameters for Function Point:

Different parameters are used to determine the function point. Those are:

1. Number of inputs: Each data item input by the user is counted.
2. Number of outputs: The outputs include reports printed, screen outputs, error message etc.
3. Number of Enquiries: It is the number of interactive queries, which are made by the user.
4. Number of files: It is the number of logical files involved.
5. Number of Interface: The interfaces are used to exchange information with other systems.

Function Point(FP):

Function point is estimated using following formula:-

FP = UFP * TCF

UFP = (No. of Inputs) * 4 + (No. of Outputs) * 5 + (No. of Enquiries)*4

+ (No. of Files) * 10 + (No. of Interfaces) * 10

TCF = DI * 0.01

UFP -> Unadjusted function point

TCF -> Technical complexity factor

DI -> Degree of Influence

The UFP reflects the functionality provided to the user.

TCF:-

The TCF refines the UFP successfully by considering 14 other factors such as transaction rate, throughput, response time etc. Each of this 14 factors assigned values from 0 to 6.

The resulting numbers are summed and known as degree of influence.

Ex:- TCF is computed as (0.65 + 0.01* DI)

DI varies from 0 to 70

So TCF varies from 0.65 to 1.35

Finally FP= UFP * TCF

Project Estimation Technique:

The estimation of project includes various project parameters. The parameters are:

- Project size
- Project duration
- Effort required to develop a software

There are 3 categories of estimation techniques

1. Empirical estimation technique
2. Heuristic estimation technique
3. Analytical estimation technique

Empirical Estimation Techniques:-

Empirical estimation technique is based on experience of project manager. The project manager guesses the project parameters based on the prior experience with the development of similar products.

Heuristic Estimation Techniques:-

Heuristic technique uses a suitable mathematical expression. Mathematical expression includes relationship among different parameters.

Analytical Estimation Techniques:-

Analytical estimation techniques derive the required result starting with certain basic assumptions. This technique does have a scientific basis. One of the analytical estimation technique is Halstead's software science and analytical estimation technique.

Empirical Estimation Technique:

It is based on experience of project management. Cost estimates can be either topdown or button-up.

In top-down estimation first focus is on costs such as computing resources, personnel required, quality assurance, system integration testing etc. In bottom-up cost estimation first estimation the cost to develop each module or subsystem. Two popular empirical estimation techniques are:

Expert Judgement Technique:

It is the most widely used cost estimation technique. In this approach an expert makes an educated guess of the problem size. But this technique is subject to human errors and individual bias.

The advantage in this technique is, when a group of expert estimates the size it minimizes the error.

Delphi Cost Estimation:-

This estimation is carried out by a team consisting of a group of experts and a coordinator. This cost estimation is carried out in following manner:

- The co-ordinator provides each estimator with a SRS document and a form for recording a cost estimate.
- Estimator study the SRS document, estimate anonymously and submit to the coordinator.
- They do not discuss their estimate with another.
- The co-ordinator prepare and distribute a summary of the estimators response.
- Based on this summary the estimator restimate.
- This process is iterated for several rounds.
- No group discussion is allowed during the entire process.

Heuristic Estimation Technique:

COCOMO:

COCOMO is Cost Constructive Model. COCOMO was proposed by Boehm. According to this model any software development complexity such as : Organic, Semi detached, Emeded.

a) Organic:

IT deals with the developing of a well understood application program. The size of development team is small.

The team members are experienced in developing similar types of project. b) Semidetached:

In semidetached mode, the development team consists of experienced and inexperienced staff. The development team is large. Requirement are not clearly known.

c) Embedded:

In the embedded mode the project has tight constraints which may related to target processor and its interface with the hardware.

According to Boehm the software cost estimation should be done through 3 stages, those are:

1. Basic COCOMO
2. Intermediate COCOMO
3. Complete COCOMO

1. Basic COCOMO:-

The basic COCOMO given by the following expressions

$= \ * (\quad) \ )$PM

$= \ * \quad) \quad h$

Where, Kloc is the estimated size of software product expressed in Kilo lines of code.

, , , are constants for each category of software product. is the estimated time

to develop the software expressed in months.

Effort is the total effort required to develop the software expressed in person month (PM).

2. Intermediate COCOMO:-

Basic COCOMO model provides weak and rough estimate but which is not accurate. Intermediate COCOMO provides estimation based on the size of the software and set of other parameters which are known as cost directive. There are fifteen cost directives which are grouped into four categories

Product Attributes:

The characteristics of product data includes the complexity of the product, reliability, requirement of product, database size etc.

Computer Attributes:

The characteristics of computer includes execution speed required, storage space required etc.

Personnel Attributes:

The attributes of development personnel are experience level, programming capability, analytical capability etc.

Development Environment:

A development environment attribute includes development facilities available to the developers.

3) Complete COCOMO (Detailed COCOMO):

The basic and intermediate COCOMO model considers a software product as a single homogeneous entity. But most large systems are made up of several smaller subsystems. The sub-systems may have different characteristics such as organic or some may be embedded or semidetached.

A complete COCOMO provides estimation of phase wise efforts and duration of phase of development.

The complete COCOMO considers different phases of the project such as software planning requirement analysis, system designing, coding etc separately. So this approach reduces the margin of error in the final estimation.

Scheduling:

Scheduling decides which task should be done and when. In order to schedule the project manager needs to do the following.

i)Identify all the tasks necessary to complete the project.

ii) Breakdown larger tasks into a logical set of small activities.

iii) Create the work breakdown structure, it determines dependency among activities and the order in which activities are executed.

iv) Decides the most likely estimates for the time duration necessary to complete the activities.

v) Resources are allocated to each activity it is done using Gantt chart.

vi) Plan the starting and ending dates of various activities. The end of each activity is called milestone.

Determine the critical path. It is the chain of activities that determine the duration of the project.

Work Breakdown Structure:

Work breakdown structure (WBS) is used to decompose a given task into small activities. The goal of the work breakdown structure is to identify all the activities that the project must undertake.

These small activities can be distributed to set of engineers which helps developing the product faster.

Two general scheduling techniques are

1) Gantt chart
2) PERT chart

1.) Gantt Chart:-

Gantt chart are used to allocate resources to activites. A Gantt chart is a special type of bar chart where each bar represent an activity. The bars are drawn against a timeline.

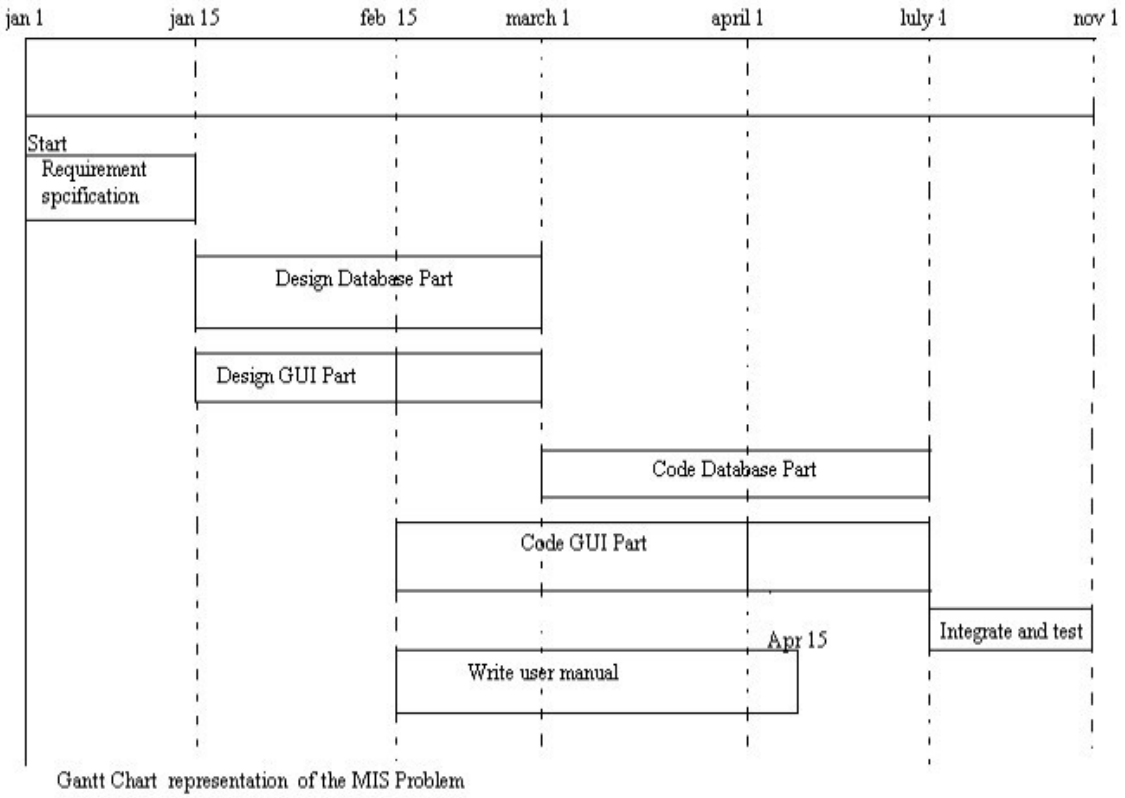


Fig. 2.2 Gantt Chart

2.) Pert Chart:-

PERT (Project Evolution & Review Technique). It contains time and cost during the project.

A PERT chart is a network of boxes and arrows. The boxes represent activities, the arrows show the dependencies of activities on one another. PERT chart is more useful for monitoring the time progress of activities.

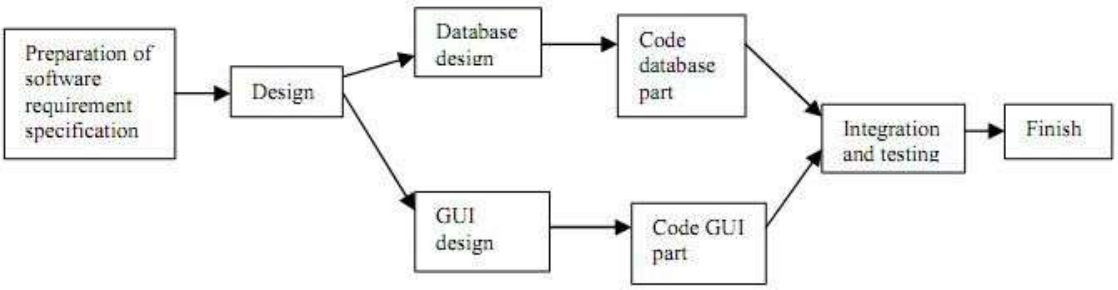


Fig. 2.3 PERT Chart

Organisation Structure:-

There are two ways in which software organisation can be structured.

1. Function format
2. Project format

In the *project format* the development staffs are divided based on the project for which they work.

In the *function format* the development staff are divided based on the functional group to which they belong to. The functional group may be design, coding, testing, maintenance etc.

Team Structure:-

It deals with individual teams organisation. Three formats of team structures are :

1. Chief Programmer
2. Democratic
3. Mixed team Organisation

1. *Chief Programmer:-*

   In this organisation (planning) a senior engineer provides the technical leadership. He/She is designated as the chief programmer. The chief programmer partitions the tasks into small activities and assigns them to the team member. The organisation is suitable for small projects.

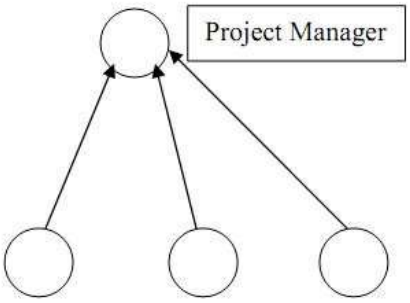


Fig.2.4 Chief Programmer Team Structure

2. *Democratic Team:-*

   In democratic team structure, there is no formal team hierarchy. The manager provides the administrative leadership. At different times, different members of the group provide technical leadership. The democratic team structure is suitable for less understood problems such as research oriented project.

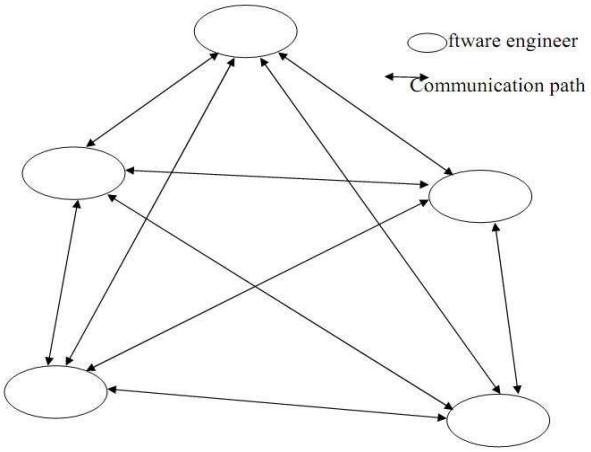


Fig 2.5 Democratic Team Structure

3. *Mixed Team Organisation:-*

The mixed team organisation uses ideas from both democratic and chief programmer organisation. It is better for large ream sizes. This is used in most of software companies.

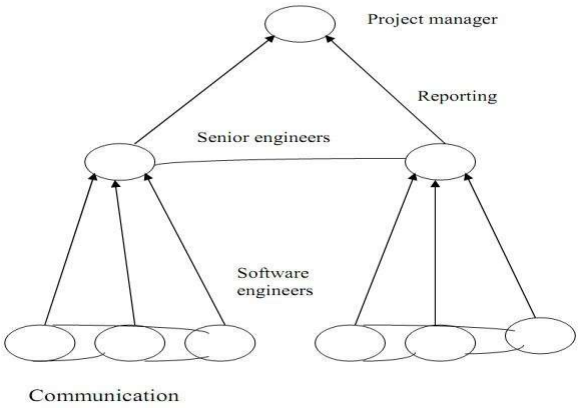


Fig. 2.6 Mixed Team Structure

Risk Management:

Risk is defined as an exposure to chance of injury or loss. It says that there is possibility that something negative may happen. In software development the risk is an adverse effect on cost, quality or schedule. Risk management consists of 3 essential activities. These are:

1. Risk Identification
2. Risk Assessment
3. Risk Containment

1. Risk Identification:-

It identifies all the different tasks for a particular project. There are three categories of risks: *a)Project Risk:*

Project risk is related to budget, scheduling personnel resources and customer related problems. As software is intangible, it is very difficult to monitor and control a software project.

b)Technical Risk:-

It deals with potential risk related to design, implementation, interfacing, testing and maintenance problem. It also includes incomplete specification, changing specification. This occurs due to development teams insufficient knowledge about the product. *c)Business Risk:-*

It includes risk of building an excellent product that no one want (no buyer). It also includes loosing budgetary or personal commitments etc.

2. **Risk Assessment:-**

   The goal of the risk assessment is to rank the risks. It helps to focus attention on more riskier items. Risk is rated in the two steps. i) Likelihood (Probability) of coming true (r) ii) Consequence of the problem associated with the risk (s)

   The priority P= r * s

3. **Risk Containment:-**

   After the identification and assessment of the risk plans must be made to contain the most damaging and most likely risk. 3 main strategies are:

   - Avoid the risk
   - It includes discussion with customer or giving initiative to engineers etc.
   - Transfer the risk:- This involve getting the risky component developed by a 3rd party or bugging insurance cover etc.

Risk reduction:- These involves planning ways to contain the damage due to risk.

QUESTIONS ON CHAPTER 3:-

2MARKS:-

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

5&7 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?

UNIT-3

REQUIREMENT ANALYSIS AND SPECIFICATION

Requirement Analysis:

The requirement analysis provides software designer with representation of system information, functional behaviour.

Software requirement analysis is divided into five parts

1. Problem recognition
2. Evaluation
3. Modeling
4. Specification
5. Review

SRS Principle:

Define the environment in which the system operates.

Establish the content and structure of a specification so that it can be changed easily.

The specification must be extensible and tolerant of incompleteness.

Create an abstract model rather than implementation model.

SRS Document:

The requirement analysis and specification phase starts after the feasibility study phase is completed.

The goal of this phase is to clearly understand the customer requirements and systematically organise these requirements into SRS document.

This phase consists of two activities such as

1. Requirement gathering and analysis
2. Requirement specification

1. Requirement Gathering & Analysis:

This activity considers two activities

a) Requirement gathering
b) Requirement analysis

a) *Requirement Gathering:* Requirement gathering activity involves interviewing the end users and customers and studying the existing documents to collect all the possible information about the system.

b) *Requirement Analysis:* Analysis of gathered document is done to clearly understand the exact requirements of the customer.

The analyst should understand what is the problem, what are the possible solutions, what are the exact inputs and outputs etc. .

After analysis the analyst tries to identify and different requirement problems. The problems are:

i) Anomaly: Anomaly is an ambiguity in the requirement when the requirement is anomalous i.e several interpretation of the requirement is possible.

ii) Inconsistency: Two requirements are said to be inconsistent if one of the requirements contradicts other. iii) Incompleteness: An incomplete set of requirement is one in which some requirements having overloaded.

2. Software Requirement Specification :

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

Users of the SRS document:

Different People need the SRS document for very different purposes. Some of the important categories of users of the SRS document and their needs are as follows.

- Users, customers and marketing personnel : The goal of this set of audience is to ensure that the system as describe in the SRS document will meet their needs.
- The software developers refer to the SRS document to make sure that they develop exactly what is required by the customer.
- Test Engineers: Their goal is to ensure that the requirements are understandable from a functionality point of view, so that they can test the software and validate it's working.
- User Documentation Writers: Their goal in reading the SRS document is to ensure that they understand the document well enough to be able to write the users' manuals.
- Project Managers : They want to ensure that they can estimate the cost of the project easily by referring to be SRS document and that it contains all information required to plan the project.
- Maintenance Engineers: The SRS document helps the maintenance engineers to understand the functionalities of the system. A clear knowledge of the functionalities can help them to understand the design and code.

Contents of the SRS Document :

An SRS document should clearly document:

- Functional Requirements
- Nonfunctional Requirements
- Goals of implementation

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.

Goals of implementation

The goals of implementation part of the SRS document gives some general suggestion regarding development. This section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future.

Characteristics of Good SRS document :

Concise: The SRS document should be concise, unambiguous, consistent and complete. Irrelevant description reduced readability and also increases error possibilities.

Structured: The SRS document should be well-structured. A well-structured document is easy to understand and modify.

Block-box View: It should specify what the system should do. The SRS document should specify the external behaviour of the system and not discuss the implementation issues. The SRS should specify the externally visible behaviour of the system.

Conceptual Integrity : The SRS document should exhibit conceptual integrity so that the reader can easily understand the contents.

Verifiable: All requirements of the system as documented in the SRS document should be verifiable if and only if there exists some finite cost-effective process with which a person of machine can check that the software meets the requirement.

Modifiable : The SRS is modifiable if and only if its structure and style are such that any changes to the requirements can be made easily, completely and consistently while retaining the structure and style.

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
Non-functional 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 Representing 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.

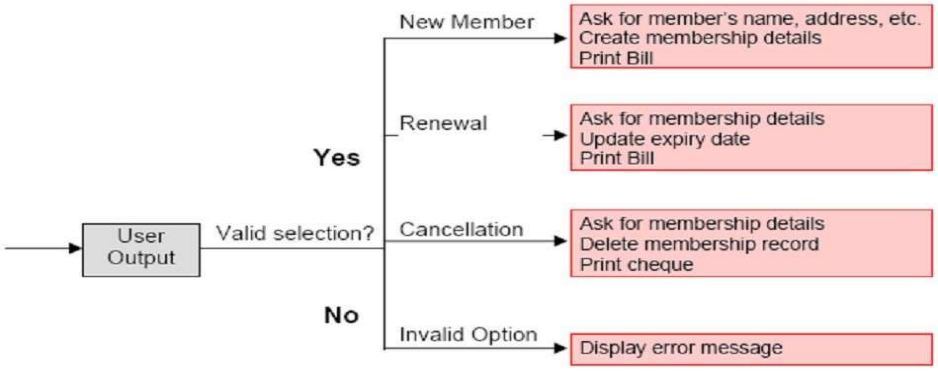


Fig. 3.1 Decision tree for LMS

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

Fig. 3.2 Decision table for LMS

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.

Questions of Chapter 3:

2MARKS:-

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

5&7 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?

UNIT-4

SOFTWARE DESIGN

Importance of Software Design

Software design aims to plan and create a blueprint for the implementation of the software. Software design transforms the SRS document into implementable form using a programming language. The following items are designed and documented during the design phase.

- Different modules in the solution should be cleanly identified. Each module should be named according to the task it performs.
- The control a relationship exists among various modules should be identified in the design document. The relationship is also known as the call relationship.
- Interface among different modules. The interface among different modules identifies the exact data items exchanged among the modules.
- Data structures of the individual modules.
- Algorithms required to implement the individual modules.

Characteristics of a good software design :

The characteristics are listed below:

Correctness: A good design should correctly implement all the functionalities of the system.

Understandability: A good design should be easily understandable.

Efficiency: A good design solution should adequately address resource, time and cost optimization issues.

Maintainability: A good design should be easy to change.

Understandability: It should assign consistent and meaningful names for various design components.

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

It should neatly arrange the modules in a hierarchy, e.g. tree-like diagram. If different modules are independent of each other, then each module can be understood separately. *Clean Decomposition:* The modules in a software design should display high cohesion and low coupling. The modules are more or less independent of each other.

Cohesion and Coupling:

The primary characteristics of a neat module decomposition are high cohesion and low coupling. A modules having high cohesion and low coupling is said to be functionally independent of other modules.

Cohesion:

Cohesion is a measure of the functional strength of a module where as the coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. A module is said to be highly cohesive if its components are strongly related to each other by some means of communication or resource sharing or the nature of responsibilities. A cohesive module performs a single task or function.

There are seven types or levels of cohesion

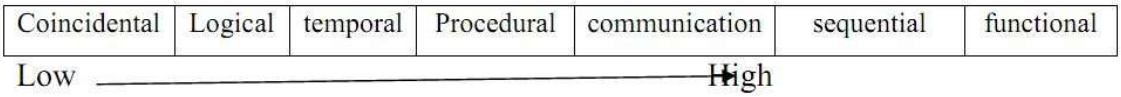


Fig. 4.1 Classification of Cohesion

Coincidental cohesion:

A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely. 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. Two modules with high coupling are strongly interconnected and thus dependent on each other.

"Uncoupled" modules have no interconnections they are completely independent. A good design will have low coupling. Coupling is measured by the number of interconnections between modules.

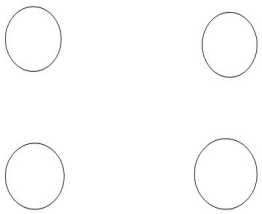


Fig. 4.2 Uncoupled modules

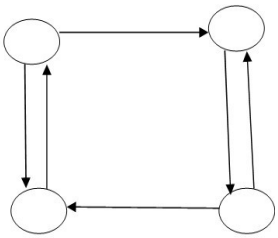


Fig. 4.3 Loosely coupled

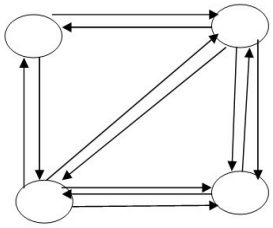


Fig. 4.4 Highly coupled

Different types of coupling are:

Data	Stamp	Control	Common	Content

Fig. 4.5 Levels of coupling

Data coupling:

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

S/W Design Approaches :

Two different approaches to software design are: Function-oriented design and Objectoriented design.

Function oriented design :

Features of the function-oriented design approach are:

Top-down decomposition :

In top-down decomposition, starting at a high-level view of the system, each highlevel function is successfully refined into more detailed functions.

Ex: Consider a function create-new-library member which essentially creates the record for a new member, assigns a unique membership number to him and prints a bill towards his membership charge. This function may consists 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.

Object Oriented Design :

In the object-oriented design approach, the system is viewed as a collection of objects. The system state is decentralized among the objects and each object manages its own state information. Objects have their own internal data which define their state. Similar objects constitute a class. Each object is a member of some class. Classes may inherit features from a super class. Conceptually, objects communicate by message passing.

Structured Analysis Methodology:

The aim of structured analysis activity is to transform a textual problem description into a graphic model. During structured design, all functions identified during structured analysis are mapped to a module structure. Structure analysis technique is based on the following principles:

Top-down decomposition approach

Divide and conquer principle. Each function is decomposed independently.

Graphical representation of the analysis results using Data Flow Diagram (DFD).

Data Flow Diagram:

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

Lists the Symbols used in DFD:

Five different types of primitive symbols used for constructing DFDs. The meaning of each symbol is

Functional symbol  *( )* : A function is represented is using a circle.

External entity  *symbol ( ) :* An external entities are essentially those physical entities external to the software system which interact with the system by inputting data to the system or by consuming the data produced by the system.

Data flow symbol  *( ) :* A directed arc or an arrow is used as a data flow symbol.

Data store  *symbol ( )* : A data store represents a logical file. It is represented using two parallel lines.

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

Construction of DFD:

A DFD start with the most abstract definition of the system (lowest level) and at each higher level DFD, more details are successively introduced. The most abstract representation of the problem is also called the context diagram.

Context Diagram:

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

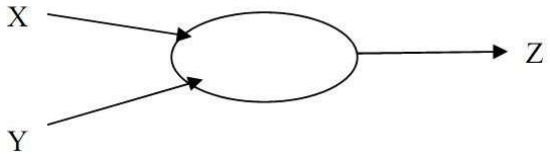


Fig. 4.6 Context diagram

Level 1 DFD :

The level 1 DFD usually contains between 3 and 7 bubbles. If there are between 3 to 7 high-level functional requirements, then these can be directly represented as bubbles in the Level 1 DFD. If a system has more than seven high-level requirements, then some of the related requirements have to be combined and represented in the form of a bubble in the Level 1 DFD.

Decomposition : Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into sub functions at the successive level of the DFD.
Decomposition of a bubble should be carried out on until a level is reached at

Example: Student admission and examination system
This statement has three modules, namely

- Registration module
- Examination module
- Result generation module *Registration module:*

An application must be registered, for which the applicant should pay the required registration fee. This fee can be paid through demand draft or cheque drawn from a nationalized bank. After successful registration an enrolment number is allotted to each student, which makes the student eligible to appear in the examination.

Examination module:

a) Assignments : Each subject has an associated assignment, which is compulsory and should be submitted by the student before a specified date.

b) Theory Papers : The theory papers can be core or elective. Core papers are compulsory papers, while in elective papers students have a choice to select.

c) Practical papers: The practical papers are compulsory and every semester has practical papers.

Result generation Module:

The result is declared on the University's website. This website contains mark sheets of the students who have appeared in the examination of the said semester.

Data Flow Diagram of Student Admission and Examination System:

Level 0 DFD:

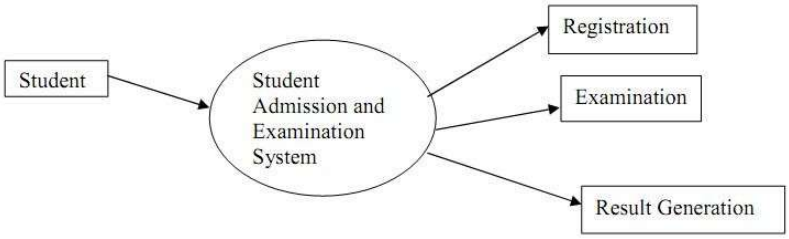


Fig.4.7 Level 0 DFD or Context Diagram

Level 1 DFD :

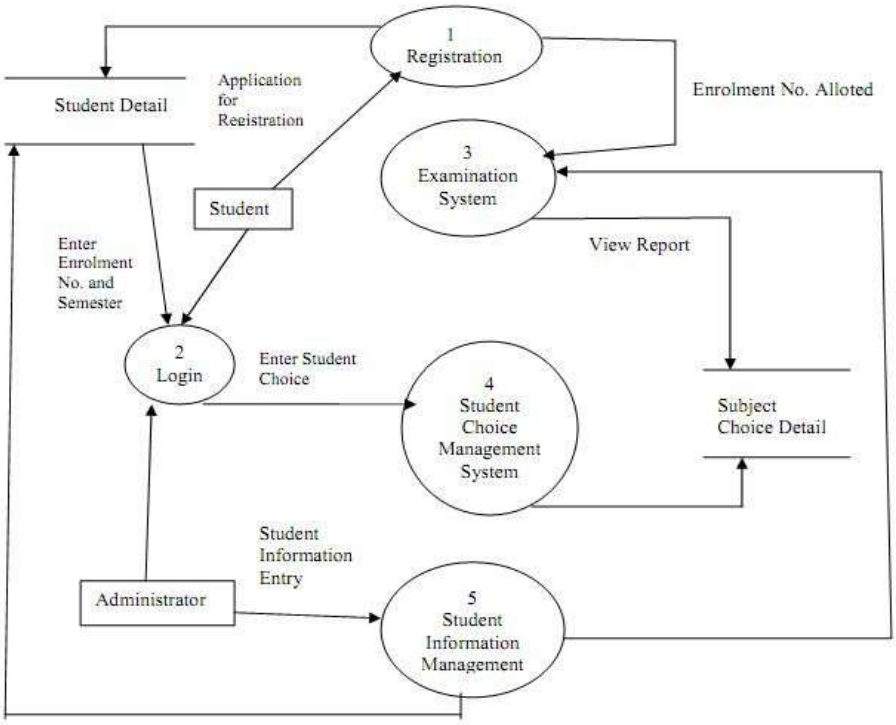


Fig. 4.8 Level 1 DFD

Level 2 DFD:

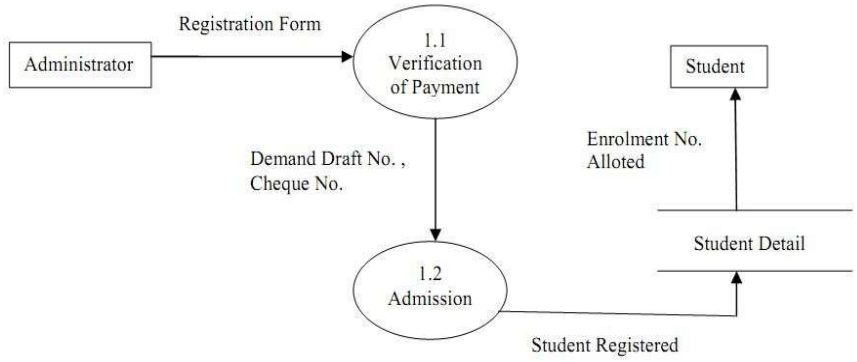


Fig. 4.9 Level 2 DFD of Registration

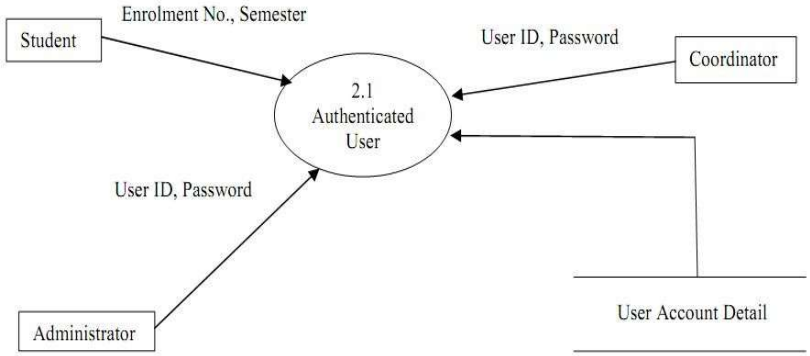


Fig. 4.10 Level 2 DFD of Authenticated

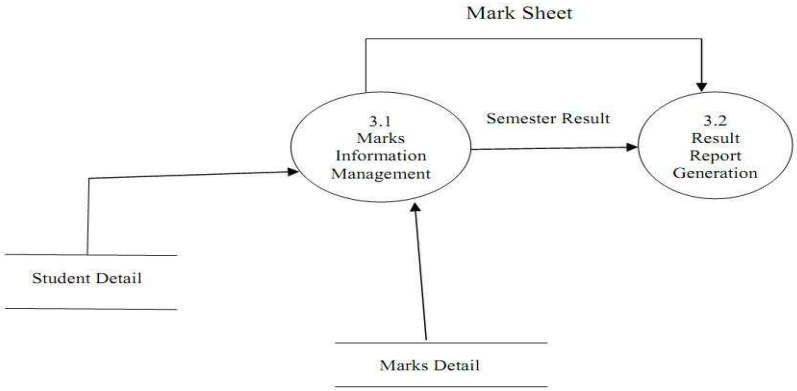


Fig 4.11Level 2 DFD of Examination

Limitations of DFD :

A data flow diagram does not show flow of control. It only shows all possible inputs and outputs for each transformation in the system.

The method of carrying out decomposition depend on the choice and judgement of the analyst. Many times it is not possible to say which DFD representation is superior or preferable to another.

The data flow diagram does not provide any specific guidance as to how exactly to decompose a given function into its sub functions.

Size of the diagram depends on the complexity of the logic.

Structured Design:

The aim of structured design is to transform the results of the structured analysis into a structure chart.

A **structure chart** represents the software architecture, i.e. the various modules making up the system, the dependency (which module calls which other modules), and the parameters that are passed among the different modules.

The structure chart representation can be easily implemented using some programming language.

The main focus in a structure chart representation is on the module structure of the software and the interactions among different modules, so how a particular functionality is achieved are not represented.

The basic building blocks which are used to design structure charts are the following:

- *Rectangular boxes:* Represents a module.
- *Module invocation arrows:* Control is passed from one module to another module in the direction of the connecting arrow.
- *Data flow arrows:* Arrows are annotated with data name; named data passes from one module to another module in the direction of the arrow.
- *Library modules:* Represented by a rectangle with double edges.
- *Selection:* Represented by a diamond symbol.
- *Repetition:* Represented by a loop around the control flow arrow.

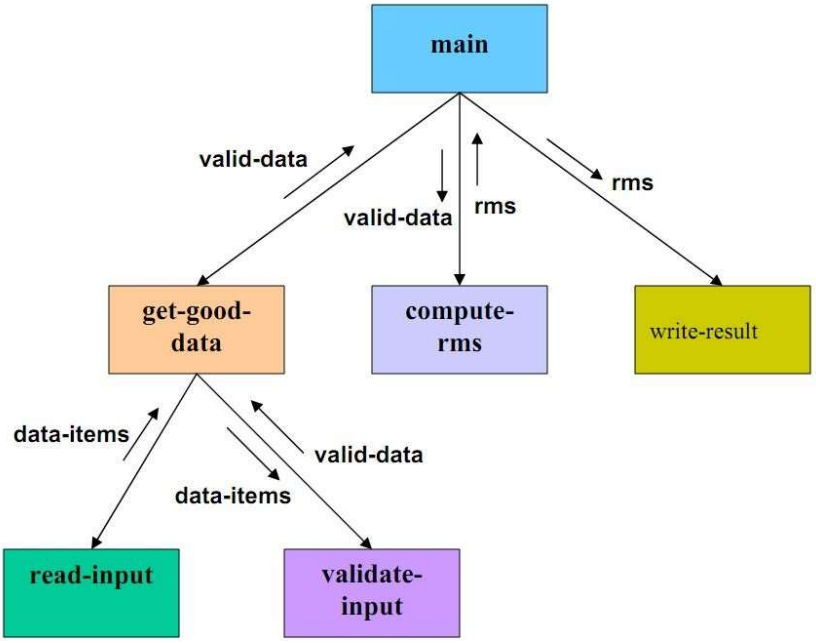


Fig. 4.12: Structure chart

Principles of transformation of DFD to Structure Chart :

Structure design provides two strategies to guide transformation of a DFD into a structure chart:

1) Transform analysis
2) Transaction analysis

Normally, one starts with the level 1 DFD, transforms it into module representation using either the transform or the transaction analysis and then proceeds towards the lower-level DFDs. At

each level of transformation, first determine whether the transform or the transaction analysis is applicable to a particular DFD.

1) Transform Analysis :

Transform analysis identifies the primary functional components (modules) and the high level input and outputs for these components.

The *first step* in transform analysis is to divide the DFD into three types of parts:

Input
Logical processing
Output

The **input portion** in the DFD includes processes that transform input data from physical to logical form. Each input portion is called an *afferent branch*.

The **output portion** of a DFD transforms output data from logical form to physical form. Each output portion is called an *efferent branch.*

The remaining portion of a DFD is called *central transform*.

In the *next step* of transform analysis, the structure chart is derived by drawing one functional component for the central transform and the afferent and efferent branches.

In the *third step* of transform analysis, the structure chart is defined by adding levels of functional components may be added. This process of breaking functional components into subcomponents is called factoring. Factoring includes adding read and write modules, errorhandling modules, initialization and termination process etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

2) Transaction Analysis:

If there are several possible paths for the DFD, then one of the possible is traversed depending upon the I/P data value. If DFD has different possible processing bubbles for different input then transaction analysis is used.

In a structure chart draw a root module and below this module, each identified transaction of a module is drawn.

Every transaction carries a tag.

Example:-

A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his/her residence address, telephone number, and the driving license number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. A customer can present his CN to the check out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded Rs.10,000. The entries against the CN are the reset on the day of every year after the prize winners' lists are generated.

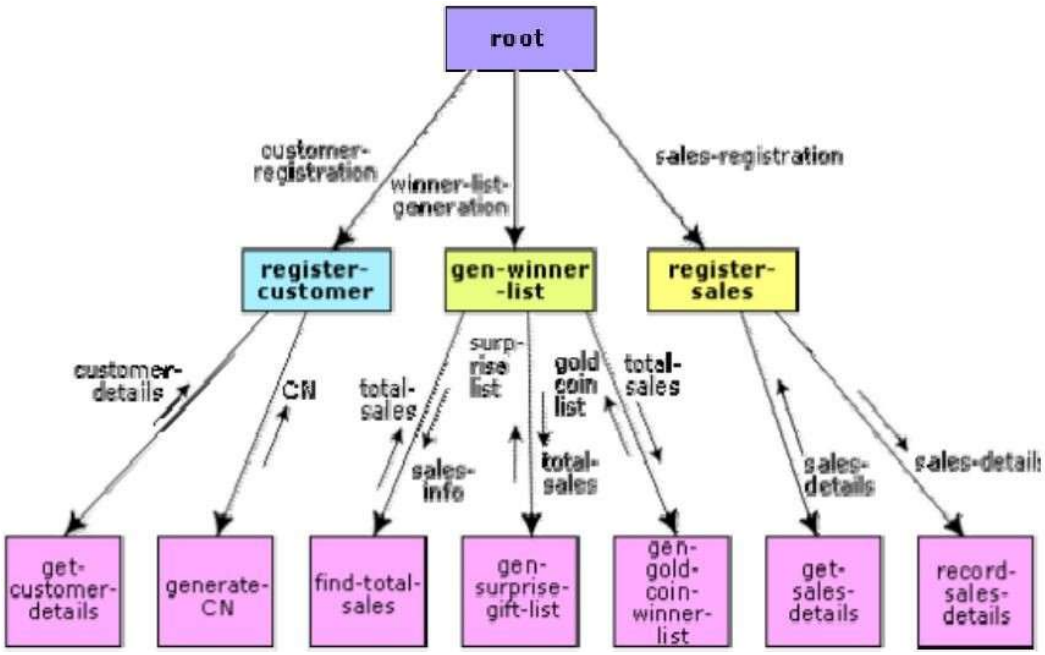


Fig. 4.13 Structure chart for the supermarket prize scheme

QUESTIONS ON 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&7 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.

UNIT-5

USER INTERFACE DESIGN

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 behaviour.

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. o 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.

o 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.

o 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.

o 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

Command language based interfaces :

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.

o A simple command language-based interface might simply assign unique names to the different commands.

o 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. o 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. rect 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, o 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

SOFTWARE CODING & TESTING

Coding Standards and Guidelines:

Good software development organizations develop their own coding standards and guidelines depending on what best suits their needs and types of products they develop. Some of the guidelines are:

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

Contents of the headers preceding codes for different modules: The information contained in the headers of different modules should be standard for an organization. Some standard header data are:

a) Name of the module
b) Date on which the module was created
c) Author's name
d) Modification history
e) Synopsis of the module

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

Error return conventions and exception handling mechanisms: The way error conditions are reported by different functions in a program and the way common exception conditions are handled should be standard within an organization.

Code Walk-Through:

The main objective of code walk-through is to discover the algorithmic and logical errors in the code. Code walkthrough is an informal code analysis technique. In this technique, after a module has been coded, it is successfully compiled and all syntax errors are eliminated. Some members of the development team are given the code a few days before the walk-through meeting to read and understand the code. Each member selects some test cases and simulates execution of the code through different statements and functions of the code. Some guidelines for code walkthrough are:

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

Code Inspection:

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

Use of uninitialized variables
Jumps into loops
Non-terminating loops
Array indicates out of bounds
Improper storage allocation and deallocation
Use of incorrect logical operators
Improper modification of loop variables

Software Documentation`

There are different kinds of documents such as user's manual, software requirements specification (SRS) document, design document, test document, installation manual are part of the software engineering process.

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

➢ Good documents help the users in effectively exploiting the system.

➢ Good documents help the manner in effectively tracking the progress of the project.
Different types of software documents can be broadly classified into:

Internal documentation

External documentation

a) Internal Documentation

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

- Comments embedded in the source code
- Use of meaningful variable names
- Module and function headers
- Code structuring (i.e. Code decomposed into modules and functions)
- Use of constant identifiers
- Use of user-defined data types

b) External documentation

External documentation is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test document etc. A systematic software development style ensures that all these documents are produced in an orderly fashion. An important feature of good documentations consistency with the code. Inconsistencies in documents creates confusion in understanding the product. Also, all the documents for a product should be up-to-date.

Testing:

A software product is normally tested in the three levels:

- Unit testing
- Integration testing
- System testing

Unit Testing:

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

Integration Testing:

An integration test is done to demonstrate that different pieces of the system work together. Integration tests cover whole applications, and they require much more effort to put together. They usually require resources like database instances and hardware to be allocated for them.

System Testing:

System tests test the entire system. It is set of test carried out by test engineer against the software(system) developed by developer. The purpose of system testing is to validate an application and to test all functions of the system. The most popular approach of system testing is Black Box testing.

Black –Box Testing :

In the black-box testing, test cases are designed from an examination of the input/output values only. There is no requirement of knowledge of design or code. Two main approaches to design black-box test cases are:

Equivalence class Partitioning
Boundary value analysis

Equivalence Class Partitioning

In this approach, the domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class.

Equivalence classes for a software can be designed by examining the input data and output data. The following are some general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.

2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for valid input values and another equivalence class for invalid input values should be defined.

Example#1: 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: The set of negative integers, the set of integers in the range of 0 and 5000, and 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: {-5,500,6000}.

Boundary Value Analysis:

A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs. 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:

White –Box testing is also known as transparent testing. It is a test case design method that uses the control structure of the procedural design to derive test cases. This testing concentrate on procedural detail.

White-box testing Methodologies:

Different white-box testing methods are:

1) **Statement coverage**

The statement coverage strategy aims to design test cases so that every statement in a program is executed at least once. The idea behind the statement coverage

strategy is that unless a statement is executed, it is very hard to determine if an error exists in that statement.

Example: Consider the Euclid's GCD computation algorithm:

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

By choosing the test set {(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.

2) Branch coverage

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

For Euclid's GCD computation algorithm , the test cases for branch coverage can be {(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)}.

3) Condition coverage

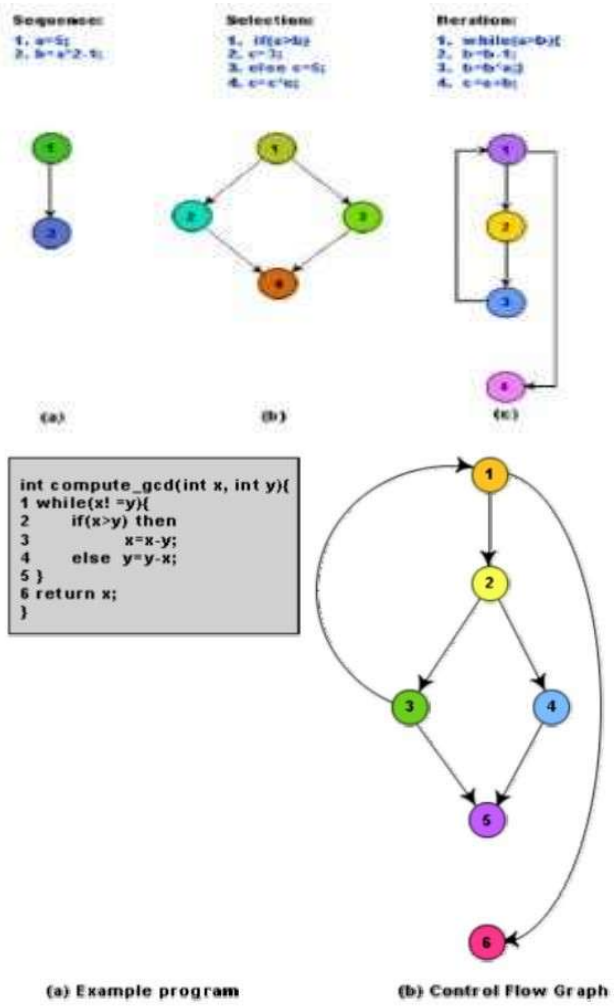
In this structural testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, in the conditional expression ((c1.and.c2).or.c3), the components c1, c2 and c3 are each made to assume both true and false values. Therefore, a condition coverage-based testing technique is practical only if n (the number of conditions) is small.

4) Path coverage

The path coverage-based testing strategy requires us to design 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.



Path

A path through a program is a node and edge sequence from the starting node to a terminal node of the control flow graph of a program. There can be more than one terminal node in a program. Writing test cases to cover all the paths of a typical program is impractical. For this reason, the path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths.

Linearly independent path

A linearly independent path is any path through the program that introduces at least one new edge that is not included in any other linearly independent paths. If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent. Thus, a path that is subpath of another path is not considered to be a linearly independent path.

Cyclomatic complexity

For more complicated programs it is not easy to determine the number of independent paths of the program. McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program.

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

Method 1:

Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:

$$V(G) = E - N + 2$$

where N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

For the CFG of example shown in fig, E=7 and N=6. Therefore, the cyclomatic complexity = 7-6+2 = 3.

Method 2:

An alternative way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows:

V(G) = Total number of bounded areas + 1

In the program's control flow graph G, any region enclosed by nodes and edges can be called as a bounded area.

For the CFG example from a visual examination of the CFG the number of bounded areas is 2. Therefore the cyclomatic complexity, computing with this method is also 2+1 = 3.

Method 3:

The cyclomatic complexity of a program can also be easily computed by computing the number of decision statements of the program. If N is the number of decision statement of a program, then the McCabe's metric is equal to N+1.

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.

1) *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.

2) *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.

3) *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. *4) 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**

Many a times, debugging requires a thorough understanding of the program design. Debugging may sometimes even require full redesign of the system.

One must be beware of the possibility that any one error correcting may introduce new errors.

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

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.

- 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:

- Bottom-up approach:

In bottom-up testing, each subsystem is tested separately and then the full system is tested.

– Testing can start only after bottom level modules are ready.

Advantages of bottom – up integration testing is that several disjoint subsystems can be tested simultaneously.

A disadvantage of bottom – up testing is the complexity occurs when the system is made up of a large number of small subsystems.

- Mixed (or Sandwiched) integration testing:

– Uses both top-down and bottom-up testing approaches.
– Most common approach.

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.

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. All performance tests can be considered as black-box tests.

- Stress testing
- Volume testing
- Configuration testing
- Compatibility testing
- Regression testing
- Recovery testing
- Maintenance testing
- Documentation testing
- Usability testing

Stress Testing

Stress testing is also known as endurance testing. Stress testing evaluated system performance when it is stressed for short periods of time. Stress tests are black – box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software.

Volume Testing

Volume testing checks whether the data structures (buffers, arrays, queues, stacks etc.) have been designed to successfully handle extraordinary situations.

Example : A compiler might be tested to check whether the symbol table overflows when a very large program is compiled.

Configuration Testing

Configuration testing is used to test system behavior in various hardware and software configuration specified in the requirements.

Compatibility Testing

This type of testing is required when the system interfaces with external systems such as databases, servers etc. Compatibility aims to checkwhether the interface functions perform as required. For instance, if the system needs to communicate with a large database system to retrieve information, compatibility testing is required to test the speed and accuracy of data retrieval.

Regression Testing

Regression testing is performed in the maintenance or development phase. This type of testing is required when the system being tested is an upgradation of an already existing system to fix some bugs or enhance functionality, performance etc.

Recovery Testing

Recovery testing tests the response of the system to the presence of faults or loss of power, devices, services data etc. For example, the printer can be disconnected to check if the system hangs. *Maintenance Testing*

Maintenance testing addresses the diagnostic programs and other procedures that are required to be developed to help implement the maintenance of the system.

Documentation Testing

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

Usability Testing

Usability testing pertains to checking the user interface to see if it meets all the user requirements. During usability testing, the display screens, messages, report formats and other aspects relating to the user interface requirements are tested.

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.

General Issues Associated with Testing

Some general issues associated with testing

i)Test documentation ii) Regression testing

Test Documentation

A piece of documentation which is generated towards the end of testing is the test summary report. The report normally covers each subsystem and represents a summary of tests which have been applied to the subsystem. It will specify how many tests have been applied to a subsystem. It will specify how many tests have been successful, how many have been unsuccessful, and the degree to which they have been unsuccessful.

Regression Testing

Regression testing does not belong to either unit testing, integration testing or system testing. Regression testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced as a result of this change made or bug fixed.

QUESTIONS ON 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&7 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.

UNIT-7

SOFTWARE RELIABILITY

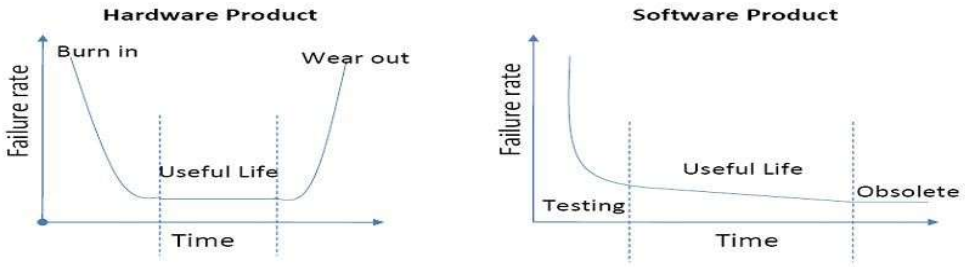
Importance of Software Reliability :

Reliability of a software product can be defined as the probability of the product working correctly over a given period of time. A software product having a large number of defects is unreliable. Different users use a software product in different ways. So defects which show up for one user may not show up for another user.

Software Reliability and Hardware Reliability :

Reliability behaviour for hardware and software is very different. Hardware failures are due to component wear and tear. If hardware failure occurs one has to either replace or repair the failed part.

A software product would continue to fail until the error is tracked down and either the design or the code is changed. For software the failure rate is highest during integration and testing phases. During the testing phase more and more errors are identified and moved resulting in a reduced failure rate. This errors removal continues at a slower speed during the useful life of the product. As the software becomes absolute, no more error correction occurs and the failure rate remains unchanged.



Different 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 specialized in the SRS document. Some reliability metrics which can be used to quantify the reliability of software products are:

1) Rate of Occurrence of Failure (ROCOF) :

ROCOF measures the frequency of occurrence of unexpected behaviour (i.e. failures). The ROCOF measure of a software product can be obtained by observing the behaviour of a software product in operation over a specified time interval and then calculating the total number of failures during this interval.

***2) Probability of Failure ON Demand (POFOD)*:**

POFOD measures the likelihood of the system failure 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. ***3) Availability :***

Availability of a system is a measure of how likely will the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (downtime) of a system when a failure occurs. In order to intimately, it is necessary to classify various types of failures. Possible classifications of failures are:

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

Permanent: Permanent failures occur for all input values while invoking a function of the system.

Recoverable: When recoverable failures occur, the system recovers with or without operator intervention.

Unrecoverable: In unrecoverable failures, the system may need to be restarted.

Cosmetics: These classes of failures cause only minor irritations, and do not lead to incorrect results.

4) Mean TIME TO Failure (MTTF)

MTTF is the average time between two successive failures, observed over a large number of failures. To measure MTTF, we can record the failure data for n failures. ***5) 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 then to fix them. ***6) Mean Time Between Failures (MTBF)***

MTBF = MTTF+MTTR

Thus, MTBF Of 300 hours indicates that once a failure occurs, the next failure is expected to occur only after 300 hours. In this case, the time measurements are real time and not the execution times as in MTTF.

Software Quality :

The objective of software engineering is to produce good quality maintainable software in time and within budget. That is a quality product does exactly what the users want it to do. The modern view of quality associates a software product with several factors such as:

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.

Correctness

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

Usability

A software product has good usability, if different categories of users can easily invoke the functions of the product.

Reusability

A software product has good reusability, if different modules of the product can easily to develop new products.

Maintainability

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

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 attain a particular level of reliability.
- Thus, reliability growth modelling can be used to determine when to stop testing to attain a given reliability level.

There are several reliability growth models but two simplest models are:

Jelinski and Moranda Model

Littlewood and Verall's Model

Jelinski and Moranda Model

The simplest reliability growth model is a step function model where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired. Such a model is shown in fig. 7.1. However, this simple model of reliability which implicitly assumes that all errors contribute equally to reliability growth, is highly unrealistic since it is already known that correction of different types of errors contribute differently to reliability growth.

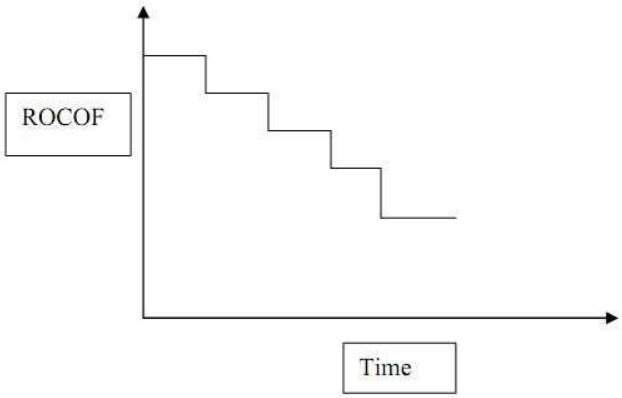


Fig. 7.1 Step function model of reliability growth

Littlewood and Verall's Model

This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors. It also models the fact that as errors are repaired, the average improvement in reliability per repair decreases (Fig. 7.2). This models the fact that error corrections with large contributions to reliability growth are removed first. This represents decreasing return as test continues.

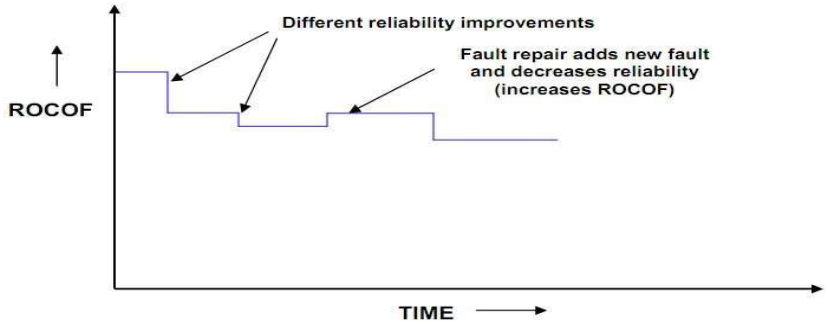


Fig. 7.2 Random-step function model of reliability growth **Software quality management system:**

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

- *Managerial Structure and Individual Responsibilities.* A quality system is actually the responsibility of the organization as a whole. However, every organization has a separate quality department to perform several quality system activities. The quality system of an organization should have support of the top management. Without support for the quality system at a high level in a company, few members of staff will take the quality system seriously.

- *Quality System Activities.* The quality system activities encompass the following:
  - auditing of projects
  - review of the quality system
  - development of standards, 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 system have rapidly evolved over the last 5 decades. The quality systems of organisation have undergone through 4-stages of evolution as :

- Quality control focuses not only on detecting the defective product & eliminating them. But also on determining the causes behind the defects.

The quality control aims at correcting the causes of errors & not just rejecting the defective products. The basic premises of modern quality assurance is that if an organizations processes are good and are followed rigorously then the products are bound to be of good quality. The modern quality paradigm includes some guidance for recognising, defining, analysing & improving the production process.

Total quality management (TQM) says that the process followed by an organisation must be continuously improve through process measurement.

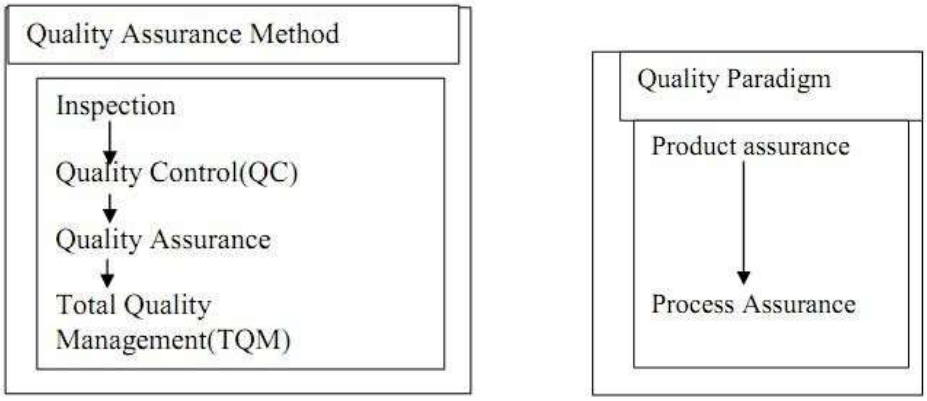


Fig. 7.3 Evolution of quality system and quality paradigm

ISO 9000 Certification :

ISO (International Standards Organization) is a consortium of 63 countries established to formulate and foster standardisation. ISO published its 9000 series of standards in 1987.

The ISO 9000 standard specifies the guidelines for maintaining a quality system. ISO 9000 specifies a set of guidelines for repeatable and high quality product development.

ISO 9000 is a series of three standards: ISO 9001, ISO 9002, and ISO 9003. *ISO 9001:* This standard applies to the organisations engaged in design, development, production, and servicing of goods. This standard is applicable to most software development organisations.

ISO 9002: This standard applies to those organisations which do not design products but are only involved in production. Examples include steel and car \ manufacturing industries. *ISO 9003:* This standard applies to organisations involved only in installation and testing of the products.

Requirement of ISO 9000 Certification :

Confidence of customers in an organisation increases when the organisation qualifies for ISO 9001 certification.

ISO 9000 requires a well-documented software production process.

ISO 9000 makes the development process focused, efficient, and cost-effective. ISO 9000 certification points out the weak points of an organization and recommends remedial action.

ISO 9000 sets the basic framework for the development of an optimal process.

Procedure to gain ISO 9000 Certification:

An organisation intending to obtain ISO 9000 certification applies to a ISO 9000 registrar for registration. The ISO 9000 registration process consists of the following stages:

1) Application: Once an organisation decides to go for ISO 9000 certification, it applies to a register for registration.
2) Pre-assessment: During this stage, the registrar makes a rough assessment of the organisation.
3) Document Review and Adequacy of Audit : During this stage, the registrar reviews the documents submitted by the organisation and makes suggestions for possible improvements.
4) Compliance audit: During this stage, the registrar checks whether the suggestions made by it during review have been complied with by the organisation or not.
5) Continued Surveillance: The registrar continues to monitor the organisation, though periodically.

SEI Capability Maturity Model (SEI CMM) :

SEI Capability Maturity Model was proposed by Software Engineering Institute of the Carnegie Mellon University, USA. SEI CMM classifies software development industries into the following five maturity levels. The different levels of SEI CMM have been designed so that it is easy for an organization to slowly build its quality system beginning from scratch. **Level 1:** *Initial*.

A software development organization at this level is characterized by ad hoc activity. Very few or no processes are defined and followed. Since software production processes are not defined, different engineers follow their own process and as a result the development efforts become chaotic. It is called chaotic level.

Level 2: *Repeatable.*

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

Level 3: *Defined*.

At this level, the processes for both management and development activities are defined and documented. There is a common organization-wide understanding of activities, roles and responsibilities. The processes though defined, the process and the product qualities are not measured. ISO 9000 aims at achieving this level.

Level 4: *Managed:*

At this level, the focus is on software metrics. Two types of metrics are collected. Product metrics measure the characteristics of the product being developed, such as its size, reliability, time complexity, understandability etc. Process metric reflect the effectiveness of the process being used, such as the average defect correction time, productivity, the average number of defects found per hour of inspection, the average number of failures detected during testing per LOC, and so forth **Level:5** *Optimizing:*

At this stage, the process and the product metrics are collected. Process and Product measurement data are analyzed for continuous process improvement.

Compare between ISO 9000 Certification and SEI/CMM:

♦ ISO 9000 is awarded by an international standards body. ISO 9000 certification can be quoted by an organization in official documents. However, SEI CMM assessment is purely for internal use.

♦ SEI CMM was specifically developed for software industry alone.

♦ SEI CMM goes beyond quality assurance and prepares on organization to ultimately achieve TQM. ISO 9000 aims at level 3 of SEI/CMM model.

QUESTIONS ON CHAPTER 7:

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.
6. Define software reliability.