

# **SKDAV GOVERNMENT COLLEGE, ROURKELA**



**DEPARTMENT OF INFORMATION TECHNOLOGY**

## **SOFTWARE ENGINEERING**

**BRANCH-IT**

**NAME OF THE FACULTY-DIPTI REKHA MISHRA**

**SEMESTER-5TH**

## **Th-3 SOFTWARE ENGINEERING**

(Common to CSE/IT)

Theory 4 Periods per week Internal Assessment 20 Marks

Total Periods 60 End Sem Exam 80 Marks

Examination 3hours Total Marks 100Marks

Topic wise distribution of periods

Sl. No. Topics	Periods
1 INTRODUCTION TO SOFTWARE ENGINEERING	06
2 SOFTWARE PROJECT MANAGEMENT	10
3 REQUIREMENT ANALYSIS AND SPECIFICATION	06
4 SOFTWARE DESIGN	10
5 USER INTERFACE DESIGN	08
6 SOFTWARE CODING & TESTING	12
7 SOFTWARE RELIABILITY	08
TOTAL	60

**RATIONALE:** Software Engineering technology is now a days largely adopted by most computer based applications to bridge the gap between a human user & the computer. By this multiple media are implemented and used in computer based application to enhance their understanding ability before a common man. This will expose the students to various project building and testing techniques which they will encounter during there professional life as a software engineer or manager.

**OBJECTIVE:** After completion of this course the student will be able to:

- Understand the concept of Software Engineering.
- Understand how costs, schedule and quality drive a software project.
- Understand the role of software process and a process model in a project.
- Understand planning and estimation of a software project.
- Understand the role of SRS in a project and how requirements are validated
- Know the key design concepts of software engineering.
- Learn the structured code inspection process.
- Learn how testing is planned and testing done.

## **1.0 Introduction to Software Engineering**

- 1.1 Program vs. Software product
- 1.2 Emergence of Software Engineering.
- 1.3 Computer Systems Engineering
- 1.4 Software Life Cycle Models
  - 1.4.1 Classical Water fall model
  - 1.4.2 Iterative Water fall model
  - 1.4.3 Prototyping model
  - 1.4.4 Evolutionary model
  - 1.4.5 Spiral model

## **2.0 Software Project Management**

- 2.1 Responsibility of Project Manager
- 2.2 Project Planning
- 2.3 Metrics for Project size estimation (LOC and FP)
- 2.4 Project Estimation Techniques
- 2.5 COCOMO Models, Basic, Intermediate and complete
- 2.6 Scheduling
- 2.7 Organization and Team structure
- 2.8 Staffing
- 2.9 Risk Management
- 2.10 Configuration Management

## **3.0 Requirement Analysis and specification**

- 3.1 Requirements gathering and analysis
- 3.2 Software Requirements Specification
  - 3.2.1 Contents of SRS
  - 3.2.2 Characteristics of Good SRS
  - 3.2.3 Organization of SRS
  - 3.2.4 Techniques for representing complexing logic

## **4.0 Software Design**

- 4.1 What is a Good S/W design
- 4.2 Cohesion and coupling
- 4.3 Neat arrangement
- 4.4 S/W Design approaches
- 4.5 Structured analysis
- 4.6 Data Flow Diagrams
- 4.7 Symbols used in DFD
- 4.8 Designing DFD
- 4.9 Developing DFD model of a system
- 4.10 Shortcomings of DFD
- 4.11 Structured design
- 4.12 Principles of transformation of DFD to Structure Chart
- 4.13 Transform analysis and Transaction Analysis
- 4.14 Design Review

## **5.0 User Interface Design**

5.1 Characteristics of Good Interface

5.2 Basic concepts of UID

5.3 Types of User interfaces

5.4 Components based GUI development

## **6.0 Software Coding & Testing**

6.1 Coding

6.2.Code Review

. 6.2.1 Code walk through

. 6.2.2 Code inspections and software Documentation

6.3 Testing

6.4 Unit testing

6.5 Black Box Testing

6.6 Equivalence class partitioning and boundary value analysis

6.7 White Box Testing

6.8 Different White Box methodologies statement coverage branch coverage, condition coverage, path coverage, cyclomatic complexity data flow based testing and mutation testing

6.9 Debugging approaches

6.10 Debugging guidelines

6.11 Integration Testing

6.12 Phased and incremental integration testing

6.13 System testing alphas beta and acceptance testing

6.14 Performance Testing, Error seeding

6.15 General issues associated with testing

## **7.0 Software Reliability**

7.1 Software Reliability

7.2 Different reliability metrics

7.3 Reliability growth modeling

7.4 Software quality

7.5 Software Quality Management System

## **BOOKS**

Sl.No	Name of Authors	Title of the Book	Name of the	publisher
01	Rajib Mall	Fundamentals of	Software Engineering	PHI
02	Deepak Jain	Software Engineering:	Principles and Practice	Oxford university
03	Jawadekar	Software Engineering:	A Primer	TM (Common to CSE/IT)

## **UNIT-1** **SOFTWARE ENGINEERING**

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

Definitions-IEEE defines software engineering as:

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.

(2) The study of approaches as in the above statement.

## **PROGRAM VBS SOFTWARE PRODUCT**

### **program**

They are usually small in size. They are lines of code or maybe 100 to 2000 lines codes on

little more.

There is no documentation or lack in documentation.

Developer himself is sole user.

Single developer or maybe 2 developers make a program.

There is no proper user interface

### **Software Product:**

Very big in size. The lines of codes are in thousands To lacs maybe more, depends on

Software Product.

Proper documentation and well documented and user manual prepared.

Large or vast number of users.

A proper and well trained team of developer indulge in development.

There is proper and full well designed user interface.

## **Emergence of Software Engineering**

Software engineering techniques have evolved over many years which resulted series of innovations and experience about writing good quality programs. Innovations and programming experiences which have contributed to the development of software engineering are briefly describe below.

### **Early Computer Programming (1950s):**

Programs were being written in assembly language.

Programs were limited to about a few hundreds of lines of assembly code.

Every programmer developed his own style of writing programs:

according to his intuition (exploratory programming).

### **High-Level Language Programming (Early 60s)**

- High-level languages such as FORTRAN, ALGOL, and COBOL were introduced:
- This reduced software development efforts greatly.
- Software development style was still exploratory.
- Typical program sizes were limited to a few thousands of lines of source code.

### **Control Flow-Based Design (late 60s)**

- Programmers found it increasingly difficult not only to write cost effective and correct programs, but also to understand and maintain programs written by others.
- Thus particular attention is paid to the design of a program's control flow structure.
- A program's control flow structure indicates the sequence in which the programs instructions are executed.

### **Data Structure-Oriented Design**

- Software engineers were now expected to develop larger more complicated software products which often required writing in excess of several tens of thousands of lines of source code. The control flow-based programs development techniques could not be satisfactorily used to handle these problems and therefore more effective program development techniques were Needed.
- Using data structure-oriented design techniques, first a program's data structures are designed. In the next step, the program design is derived from the data structure.

### **Object-Oriented Design**

- An object-Oriented design technique is an intuitively appealing approach, where the natural objects occurring in a problem are first identified and then the relationships the objects such as composition, reference, and inheritance are determined. Each object essentially acts as a data hiding or data Abstraction entry.
- Object-oriented techniques have gained wide acceptance because of their simplicity, code and design reuse scope they offer and promise of lower development time, lower development cost more robust code and easier maintenance.

## **SOFTWARE LIFE CYCLE MODELS**



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

The software development life cycle (SDLC) is a framework defining tasks performed at each step in the software development process. SDLC is a structure followed by a development team within the software organization. It consists of a detailed plan describing how to develop, maintain and replace specific software. The life cycle defines a methodology for improving the quality of software and the overall development process.

### **Feasibility Study**

A feasibility study is simply an assessment of the practicality of a proposed plan or project. As the name implies, these studies ask: Is this project feasible or not. Do we have the people, tools, technology, and resources necessary for this project to succeed? Will the project get us the [return on investment](#) (ROI) that we need and expect?

The goals of feasibility studies are as follows:

1. To understand thoroughly all aspects of a project, concept, or plan
2. To become aware of any potential problems that could occur while implementing the project
3. To determine if, after considering all significant factors, the project is viable—that is, worth undertaking

### **Requirement Analysis**

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

Business analyst and Project Manager set up a meeting with the customer to gather all the information like what the customer wants to build, who will be the end user, what is the purpose of the product. Before building a product a core understanding or knowledge of the product is very important.

### **Design**

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

### **Implementation or Coding**

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

### Testing

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

Retesting, regression testing is done till the point at which the software is as per the customer's expectation. Testers refer SRS document to make sure that the software is as per the customer's standard.

### Deployment

Once the product is tested, it is deployed in the production environment or first [UAT \(User Acceptance testing\)](#) is done depending on the customer expectation.

In the case of UAT, a replica of the production environment is created and the customer along with the developers does the testing. If the customer finds the application as expected, then sign off is provided by the customer to go live.

### Maintenance

After the deployment of a product on the production environment, maintenance of the product i.e. if any issue comes up and needs to be fixed or any enhancement is to be done is taken care by the developers.

## **CLASSICAL WATERFALL MODEL AND ITERATIVE WATERFALL MODEL**

This model is called as linear sequential model. This model suggests a systematic approach to software development. The project development is divided into sequence of well-defined phases. The classical waterfall model breaks down the life cycle into an intuitive set of phases. Different phases of this model are:

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

### **Feasibility Study**

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

- **Technical Feasibility**

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

- **Economic Feasibility**

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

- **Operational Feasibility**

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

## **Requirement Analysis and Specifications**

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

This phase consists of two distinct activities:

- ☐ Requirements gathering and analysis.
- ☐ Requirements specification.

### **Requirements Gathering and Analysis**

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

### **Requirements Specification**

The customer requirements identified during the requirement gathering and analysis activity are organized into a software requirement specification (SRS) document. The requirements describe the “what” of a system, not the “how”. This document written in a natural language contains a description of What the system will do without describing how it will be done. The most important contents of this document are the functional requirements, the non functional requirements and the goal of implementation. SRS document may act as a contract between the development team and customer.

### **Design**

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

- Traditional design approach
- Object-oriented design approach

### **Traditional Design Approach**

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

1. first a structured analysis of the requirements specification is carried out,
2. second structured design activity.
  1. Structured analysis involves preparing a detailed analysis of the different functions to be supported by the system and identification of the data flow among the functions.
  2. Structured design consists of two main activities:  
Architectural design (also called high level design) and detailed design (also called low level design).

### **Object-Oriented Design Approach**

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

### **Coding and Unit Testing**

The purpose of the coding and unit testing phase of software development is to translate the software design into source code. During testing the major activities are centred on the examination and modification of the code. Initially small units are tested in isolation from rest of the software product.

### **Integration and System Testing**

During the integration and system testing phase the different modules are integrated in a planned manner. Integration of various modules are normally carried out incrementally over a number of steps. System testing usually consists of three different kinds of testing activities:

- $\alpha$  -testing:  $\alpha$  testing is the system testing performed by the development team.
- $\beta$  -testing: This is the system testing performed by a friendly set of customers.
- Acceptance testing: This is the system testing performed by the customer himself after the product delivery to determine whether to accept the delivered product or to reject it.

### **Maintenance**

Software maintenance is a very broad activity that includes error correction, enhancement of capabilities and optimization. The purpose of this phase is to preserve the value of the software over time. Maintenance involves performing the following activities:

- **Corrective Maintenance**

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

- **Perfective Maintenance**

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

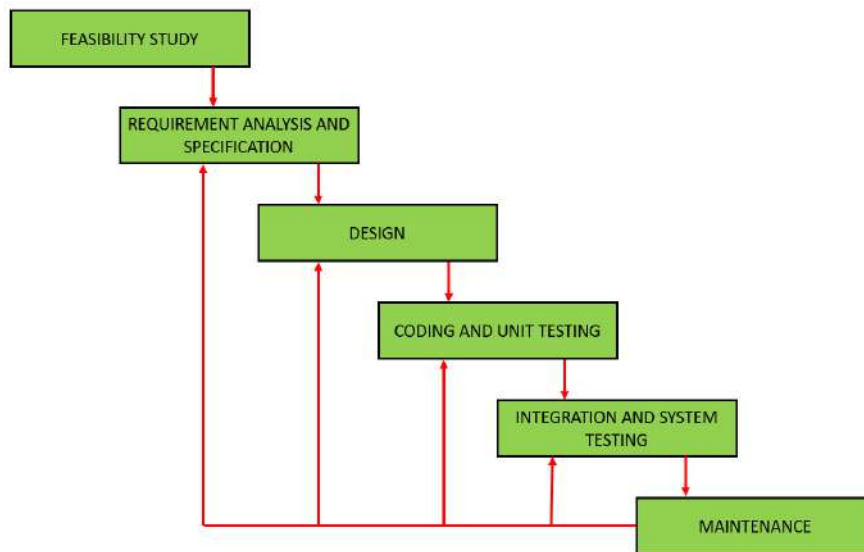
- **Adaptive Maintenance**

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

## **ITERATIVE WATERFALL MODEL**

In a practical software development project, the [classical waterfall model](#) is hard to use. So, Iterative waterfall model can be thought of as incorporating the necessary changes to the classical waterfall model to make it usable in practical software development projects. It is almost same as the classical waterfall model except some changes are made to increase the efficiency of the software development.

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



When errors are detected at some later phase, these feedback paths allow correcting errors committed by programmers during some phase. The feedback paths allow the phase to be reworked in which errors are committed and these changes are reflected in the later phases. But, there is no feedback path to the stage – feasibility study, because once a project has been taken, does not give up the project easily. It is good to detect errors in the same phase in which they are committed. It reduces the effort and time required to correct the errors.

**Phase Containment of Errors:** The principle of detecting errors as close to their points of commitment as possible is known as Phase containment of errors.

### **Advantages of Iterative Waterfall Model**

- **Feedback Path:** In the classical waterfall model, there are no feedback paths, so there is no mechanism for error correction. But in iterative waterfall model feedback path from one phase to its preceding phase allows correcting the errors that are committed and these changes are reflected in the later phases.
- **Simple:** Iterative waterfall model is very simple to understand and use. That's why it is one of the most widely used software development models.

### **EVOLUTIONARY MODEL**

**Evolutionary model** is a combination of [Iterative](#) and [Incremental model](#) of software development life cycle. It is better for software products that have their feature sets redefined during development because of user feedback and other factors. The Evolutionary development model divides the development cycle into smaller, incremental waterfall models in which users are able to get access to the product at the end of each cycle.

Feedback is provided by the users on the product for the planning stage of the next cycle and the development team responds, often by changing the product, plan or process. Therefore, the software product evolves with time. Evolutionary model suggests breaking down of work into smaller chunks, prioritizing them and then delivering those chunks to the customer one by one. The number of chunks is huge and is the number of deliveries made to the customer. The main advantage is that the customer's confidence increases as he constantly gets quantifiable goods or services from the beginning of the project to verify and validate his requirements. The model allows for changing requirements as well as all work is broken down into maintainable work chunks.

### **Application of Evolutionary Model:**

1. It is used in large projects where you can easily find modules for incremental implementation. Evolutionary model is commonly used when the customer wants to start using the core features instead of waiting for the full software.
2. Evolutionary model is also used in object oriented software development because the system can be easily portioned into units in terms of objects.

**Advantages:**

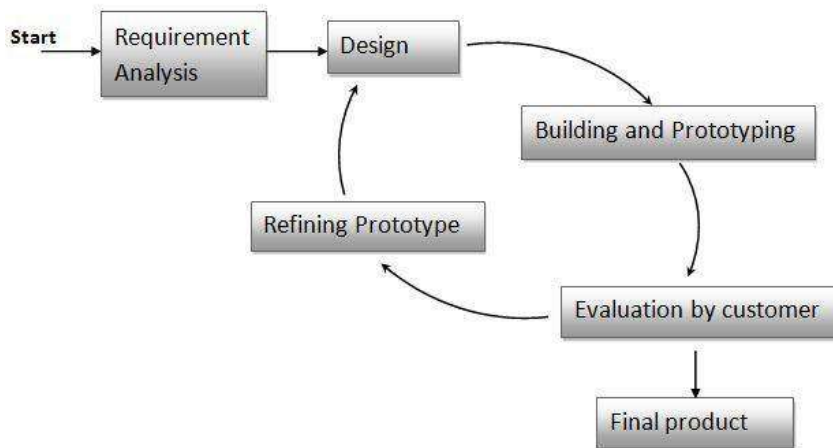
- In evolutionary model, a user gets a chance to experiment partially developed system.
- It reduces the error because the core modules get tested thoroughly.

**PROTOTYPING MODEL**

Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help to determine the requirements.

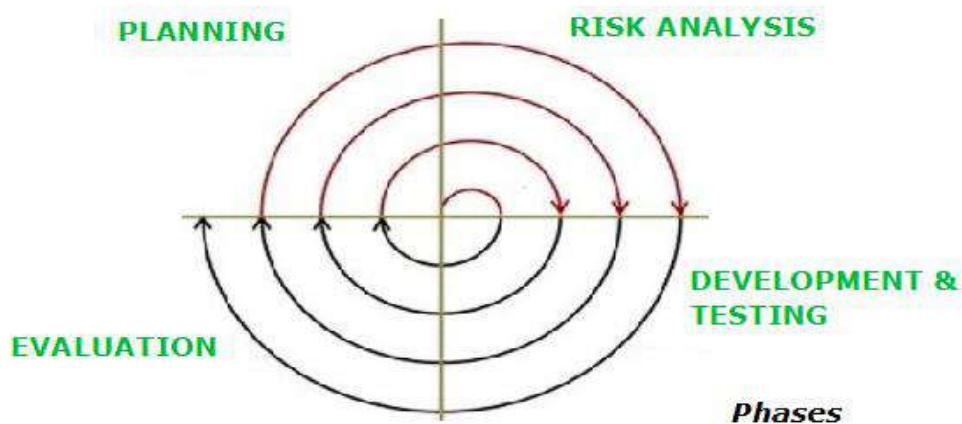
The main principle of prototyping model is that the project is built quickly to demonstrate the customer who can give more inputs and feedback. This model will be chosen

- When the customer defines a set of general objectives for software but does not provide detailed input, processing or output requirements.
- Developer is unsure about the efficiency of an algorithm or the new technology is applied.
- The development of the prototype starts when the preliminary version of the requirements specification document has been developed. A quick design is carried out and the prototype is built.
- The developed prototype is submitted to the customer for his evaluation. Based on the experience, they provide feedback to the developers regarding the prototype: what is correct, what needs to be modified, what is missing, what is not needed etc.
- Based on the customer feedback the prototype is modified and then the users and the clients are again allowed to use the system.
- This cycle of obtaining customer feedback and modifying the prototype continues till the customer approves the prototype.
- After the finalization of software requirement and specification (SRS) document, the prototype is discarded and actual system is then developed using the iterative waterfall approach.



### **Spiral Model**

1. Spiral Model is one of the oldest form of the [Software Development Life Cycle Models\(SDLC\)](#), which was first defined by the [Barry Boehm](#) in the year 1986.
2. Basically, this model is an evolutionary type model, which works on the combined approach of the [waterfall](#) and [iterative model](#).
3. Spiral model, generally involves four phases, which are followed repeatedly, in each round of the model, until no further requirements are needed to be implemented in the software product.



Phases of Spiral Model:



## 1. Planning:

This phase, mainly consists of following activities:

- Gathering of requirements through consistent interaction with the client and stakeholders.
- Feasibility study.
- Study and analysis of these requirements, to estimate the budget, resource, time, etc., required in the software development.

## 2. Risk Analysis:-

This is the crucial stage and needs to be carried out attentively, in order to identify all the potential risks, associated with the software product, and accordingly, designing and preparing the risk strategy and mitigation plan.

## 3. Development & Test:-

It is an important phase of this model, where all the requirements, strategies and plans are implemented and executed, so as to develop the software product. Further, the software development process is subsequently followed by [testing activities](#), where the developed software product is being evaluated, to assess its accurate working. Accordingly, reports are generated, which includes defects identified along with the possible solution.

## 4. Evaluation:-

This phase involves the software product interaction with the customers, who assess them and accordingly, provide their feedbacks, which helps in determining the requirements or features that needs to be added or removed from the software, in the next iteration, so as to satisfy the customer's need.

## **Spiral Model Strengths**

1. Provides early indication of risks, without much cost.

2. Critical high-risk functions are developed first.
3. Early and frequent feedback from users.
4. Cumulative costs assessed frequently.

### **Spiral Model Weaknesses**

1. The model is complex.
2. Risk assessment expertise is required.
3. May be hard to define objectives.
4. Spiral may continue indefinitely.
5. Time spent planning, resetting objectives, doing risk analysis and

## **UNIT -II**

### **SOFTWARE PROJECT MANAGEMENT**

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

The management of software development is dependent on four factors:

- The People
- The Product
- The Process
- The Project

#### **JOB RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER**

- Software managers are responsible for planning and scheduling project development. Manager must decide what objectives are to be achieved, what resources are required to achieve the objectives, how and when the resources are to be acquired and how the goals are to be achieved.
- Software managers take responsibility for project proposal writing, project cost estimation, project staffing, project monitoring and control, software configuration management, risk management, interfacing with clients, managerial report writing and presentation.
- Software managers monitor progress to check that the development is on time and within budget.

#### **SKILLS NECESSARY FOR SOFTWARE PROJECT MANAGEMENT**

- Good qualitative judgment and decision-making capabilities
- Good knowledge of latest software project management techniques such as cost estimation, risk management, configuration management.
- Good communication skill and previous experience in managing similar projects.

#### **PROJECT PLANNING**

Software managers are responsible for planning and scheduling project development. They monitor progress to check that the development is on time and within budget. The first component of software engineering project management is

effective planning of the development of the software. Project planning consists of the following activities:

- Estimate the size of the project.
- Estimate the cost and duration of the project. Cost and duration estimation is usually based on the size of the project. Estimate how much effort would be required?
- Staff organization and staffing plans.
- Scheduling man power and other resources.
- The amount of computing resources (e.g. workstations, personal computers and database software). Resource requirements are estimated on the basis of cost and development time.
- Risk identification, analysis.
- Size estimation is the first activity. The size is the key parameter for the estimation of other activities.
- Other components of project planning are estimation of effort, cost, resources and project duration. Size Estimation Effort Estimation Cost Estimation Duration Estimation Project Staffing Scheduling Precedence Ordering among Planning Activities

### **METRICS FOR PROJECT SIZE ESTIMATION**

- It's important to understand that project size estimation is the most fundamental parameter. If this is estimated accurately then all other parameters like effort, duration, cost, etc can be determined easily.
- The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.

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

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

### **LINES OF CODE**

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

### **Shortcomings of LOC**

- LOC is language dependent. A line of assembler is not the same as a line of COBOL.
- LOC measure correlates poorly with the quality and efficiency of the code.
- A larger code size does not necessary imply better quality or higher efficiency.
- LOC metrics penalizes use of higher level programming languages, code reuse etc.
- It is very difficult to accurately estimate LOC in the final product from the problem specification. The LOC count can be accurately computed only after the code has been fully developed

### **FUNCTION POINT METRICS**

- Function point metrics overcomes many of the shortcomings of LOC. Function point metrics proposes that size of the software project is directly dependent on various functionalities it supports. More the features supported the more would be the size.
- This technique helps determine size of the project directly from the problem specification so is really helpful to project managers during project planning while determining size.
- Function point metric estimates the size of a software product directly from the problem specification.

The different parameters are:

- **Number Of Inputs:**

Each data item input by the user is counted.

- **Number Of Outputs:**

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

- **Number Of Inquiries:**

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

- **Number Of Files:**

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

- **Number Of Interfaces:**

Here the interfaces which are used to exchange information with other systems. Examples of interfaces are data files on tapes, disks, communication links with other systems etc.

Function Point (FP) is estimated using the formula:

**FP = UFP (Unadjusted Function Point) \* TCF (Technical Complexity Factor)**

**UFP = (Number of inputs) \* 4 + (Number of outputs) \* 5 + (Number of inquiries) \* 4 + (Number of files) \* 10 + Number of interfaces) \* 10**

**TCF = DI (Degree of Influence) \* 0.01**

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

**Example-** Once the unadjusted function point (UFP) is computed, the technical complexity factor (TCF) is computed next. The TCF refines the UFP measure by considering fourteen other factors such as high transaction rates, throughput and response time requirements etc. Each of these 14 factors is assigned a value from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). Now, the TCF is computed as  $(0.65 + 0.01 * DI)$ .

As DI can vary from 0 to 70, the TCF can vary from 0.65 to 1.35.

Finally  $FP = UFP * TCF$

### **Feature Point Metric**

Feature point metric incorporates an extra parameter in to algorithm complexity. This parameter ensures that the computed size using the feature point metric reflects the fact that the more the complexity of a function, the greater the effort required to develop it and therefore its size should be larger compared to simpler functions.

### **PROJECT ESTIMATION TECHNIQUES**

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

- Project size(i.e. size estimation)
- Project duration
- Effort required to develop the software

There are three broad categories of estimation techniques:

- Empirical estimation techniques
- Heuristic techniques
- Analytical estimation techniques

- **EMPIRICAL ESTIMATION TECHNIQUES**

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

## **EXPERT JUDGMENT TECHNIQUE**

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

However, this technique is subject to human errors and individual bias. An expert making an estimate may not have experience and knowledge of all aspects of a project. The advantage of expert judgment is the estimation made by a group of experts. Estimation by a group of experts minimizes factors such as lack of familiarity with a particular aspect of a project, personal bias.

## **DELPHI COST ESTIMATION**

Delphi cost estimation approach tries to overcome some of the short comings of the expert judgment approach. Delphi estimation is carried out by a team consisting of a group of experts and a coordinator. The Delphi technique can be adapted to software cost estimation in the following manner:

- A coordinator provides each estimator with the software requirement specification (SRS) document and a form for recording a cost estimate.
- Estimators study the definition and complete their estimates anonymously and submit it to the coordinator. They may ask questions to the coordinator, but they do not discuss their estimates with one another.
- The coordinator prepares and distributes a summary of the estimator's responses and includes any unusual rationales noted by the estimators.
- Based on this summary, the estimators re-estimate. This process is iterated for several rounds. No group discussion is allowed during the entire process.

## **HEURISTIC TECHNIQUES**

Heuristic techniques assume that the relationships among the different project parameters can be modelled using suitable mathematical expressions. Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the value of the basic parameters in the mathematical expression.

Different heuristic estimation models can be divided into two categories:

- Single variable model
- Multivariable model

A single variable estimation model takes the following form:

Estimated parameter =  $c_1 \cdot e_1$

Where  $e$  is a characteristics of the software,  $c_1$  and  $d_1$  are constants.

A multivariable cost estimation model takes the following form:

Estimated Resource =  $c_1 \cdot e_1$

$d_1 + c_2 \cdot e_2$

$d_2 + \dots$

Where  $e_1, e_2 \dots$  are the basic characteristics of the software.

$c_1, c_2, d_1, d_2 \dots$  are constants.

### **COCOMO MODEL**

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

**ORGANIC,**

**SEMIDETACHED,**

**EMBEDDED.**

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

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

### **Basic COCOMO**

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

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



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

Where

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

Estimation of development effort:

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

Organic: **Effort = 2.4(KLOC)<sup>1.05</sup> PM**

Semi-Detached: **Effort = 3.0(KLOC)<sup>1.12</sup> PM**

Embedded: **Effort = 3.6(KLOC)<sup>1.20</sup> PM**

**PM: Person Months**

**Estimation of development time:**

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

Organic:  **$T_{dev} = 2.5(\text{Effort})^{0.38}$  Months**

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

Embedded:  **$T_{dev} = 2.5(\text{Effort})^{0.32}$  Months**

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

**Solution:** The basic COCOMO equation takes the form:

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

$$T_{dev} = b_1 \times (\text{efforts})^{b_2} \text{ Months}$$

$$\text{Estimated Size of project} = 400 \text{ KLOC}$$

### **(i)Organic Mode**

$$E = 2.4 * (400)^{1.05} = 1295.31 \text{ PM}$$
$$D = 2.5 * (1295.31)^{0.38} = 38.07 \text{ PM}$$

### **(ii)Semidetached Mode**

$$E = 3.0 * (400)^{1.12} = 2462.79 \text{ PM}$$
$$D = 2.5 * (2462.79)^{0.35} = 38.45 \text{ PM}$$

### **(iii) Embedded Mode**

$$E = 3.6 * (400)^{1.20} = 4772.81 \text{ PM}$$
$$D = 2.5 * (4772.8)^{0.32} = 38 \text{ PM}$$

## **Intermediate COCOMO**

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

the software and a set of other parameters known as cost directives. This model makes computations on the basis of 15 cost drivers based on the various attributes of software development. Cost drivers are used to adjust the nominal cost of a project to the actual project environment, hence increasing the accuracy of the estimate.

The cost drivers are grouped into four categories:

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

### **Product**

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

### **Computer**

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

## **Personnel**

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

## **Development Environment**

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

## **Complete COCOMO / Detailed COCOMO**

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

Some subsystem may be considered organic type, some embedded and some semidetached. Software development is executed in different phases and hence the estimation of efforts and schedule of deliveries should be carried out phase wise. Detailed COCOMO provides estimated phase-wise efforts and duration of phase of development. Detailed COCOMO classifies the organic, semidetached, and embedded project further into small, intermediate, medium and large-size projects based on the size of the software measured in KLOC. Based on this classification, the percentage of efforts and schedule have been allocated for different phase of the project, viz. software planning, requirement analysis, system designing, detailed designing, coding, unit testing, integration and system testing. Total effort is estimated separately. This approach reduces the margin of error in the final estimate.

## **ANALYTICAL ESTIMATION TECHNIQUES**

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

### **Halstead's Software Science an Analytical Estimation Techniques**

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

- $\eta_1$  be the number of unique operators used in the program
- $\eta_2$  be the number of unique operands used in the program
- $N_1$  be the total number of operators used in the program
- $N_2$  be the total number of operands used in the program.

There is no general agreement among researchers on what is the most meaningful way to define the operators and operands for different programming languages. For instance, assignment, arithmetic, and logical operators are usually counted as operators. A pair of parentheses, as well as a block begin and block end pair, are considered as single operators. The constructs `if.....then.....else.....endif` and a `while.....do` are treated as single operators. A sequence operator `;` is treated as a single operator.

### **Operators and Operands for the ANSI C Language**

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

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

### **Length and Vocabulary**

The length of a program as defined by Halstead, quantifies the total usage of all operations and operands in the program. Thus, length  $N = N_1 + N_2$

The program vocabulary is the number of unique operators and operands used in the program. Thus, program vocabulary  $\eta = \eta_1 + \eta_2$

### **Program Volume**

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

$$V = N \log_2 \eta$$

The program volume  $V$  is the minimum number of bits needed to encode the program. In fact, to represent  $\eta$  different identifiers uniquely, we need at least  $\log_2 \eta$  bits. We need  $N \log_2 \eta$  bits to store a program of length  $N$ . Therefore, the volume  $V$  represents the size of the program by approximately compensating for the effect of the programming language used.

### **Effort and Time**

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

### **Actual Length Estimation**

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

$$N = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

## Project Scheduling

Project-task scheduling is a significant project planning activity. It comprises deciding which functions would be taken up when. To schedule the project plan, a software project manager wants to do the following:

1. Identify all the functions required to complete the project.
2. Break down large functions into small activities.
3. Determine the dependency among various activities.
4. Establish the most likely size for the time duration required to complete the activities.
5. Allocate resources to activities.
6. Plan the beginning and ending dates for different activities.
7. Determine the critical path. A critical way is the group of activities that decide the duration of the project.

### **Work Breakdown Structure**

Most project control techniques are based on breaking down the goal of the project into several intermediate goals. Each intermediate goal can be broken down further. This process can be repeated until each goal is small enough to be well understood. Work breakdown structure (WBS) is used to decompose a given task set recursively into small activities. In this technique, one builds a tree whose root is labeled by the problem name. Each node of the tree can be broken down into smaller components that are designated the children of the node. This “work breakdown” can be repeated until each leaf node in the tree is small enough to allow the manager to estimate its size, difficulty and resource requirements. The goal of a work breakdown structure is to identify all the activities that a project must undertake.

### **Activity Networks and Critical Path Method**

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

Managers can estimate the time duration for the different tasks in several ways. A path from the start node to the finish node containing only critical tasks is called a critical path.

## Critical Path Method

□□ From the activity network representation, the following analysis can be made:

- The minimum time (MT) to complete the project is the maximum of all paths from start to finish.
- The earliest start (ES) time of a task is the maximum of all paths from the start to this task.
- The latest start (LS) time is the difference between MT and the maximum of all paths from this task to the finish.
- The earliest finish time (EF) of a task is the sum of the earliest start time of the task and the duration of the task.
- The latest finish (LF) time of a task can be obtained by subtracting maximum of all paths from this task to finish from MT.
- The slack time (ST) is  $LS - EF$  and equivalently can be written as  $LF - EF$ . The slack time is the total time for which a task may be delayed

before it would affect the finish time of the project. The slack time indicates the flexibility in starting and completion of tasks. A critical task is one with a zero slack time.

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

□□ The above parameters for different tasks for the MIS problem ( are shown in the following table.

Task	ES	EF	LS	LF	ST
Specification Part	0	15	0	15	0
Design Database Part	15	60	15	60	0
Design GUI Part	15	45	90	120	75

Code Database Part	60	165	60	165	0
Code GUI Part	45	90	120	165	75
Integrate and Test	165	285	165	285	0
White User Manual	15	75	225	285	210

The critical paths are all the paths whose duration equals MT.

### **GANTT CHART**

A Gantt chart, commonly used in project management, is one of the most popular and useful ways of showing activities (tasks or events) displayed against time. On the left of the chart is a list of the activities and along the top is a suitable time scale. Each activity is represented by a bar; the position and length of the bar reflects the start date, duration and end date of the activity. This allows you to see at a glance:

- What the various activities are
- When each activity begins and ends
- How long each activity is scheduled to last
- Where activities overlap with other activities, and by how much
- The start and end date of the whole project

To summarize, a Gantt chart shows you what has to be done (the activities) and when (the schedule).



A simple Gantt chart

## **Organization Structure**

### **Organization structure:**

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

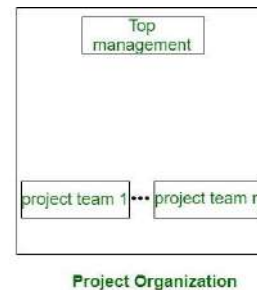
There are basically 2 broad ways in which a software package development organization is structured: *Project format*, and *Functional format*. These are explained as following below.

## 1. Project format:

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

Thus, the identical team carries out all the life cycle activities. Obviously, the functional format needs a lot of communication among groups than the project

format, as a result of one team should perceive the work done by the previous groups.

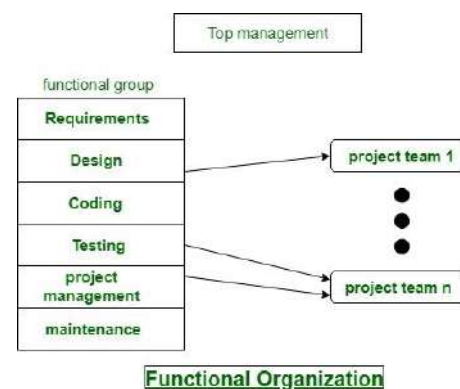


## 2. Functional format:

In the functional format, totally different groups of programmers perform different phases of a project. For example, one team may do the necessities specification, another do the planning, and so on. The partially completed product passes from one team to a different because the project evolves.

Therefore, the useful format needs significant communication among the various groups as a result of the work of 1 team should be clearly understood by the next teams engaged on the

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



## Team Structure

Team structures address the issue of organization of the individual teams.



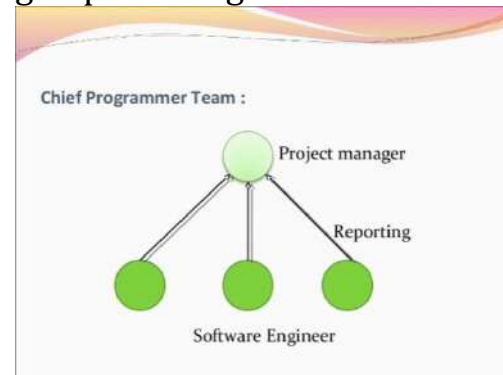
- ☐ ☐ Chief programmer
- ☐ ☐ Democratic
- ☐ ☐ Mixed team organization

### Chief Programmer Team

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

The chief programmer provides an authority. The chief programmer team leads to lower team morale, since the team member's work under the constant supervision of the chief programmer. This also inhibits their original thinking.

The chief programmer team is probably the most efficient way of completing and small projects. The chief programmer team structure works well when the task is within the intellectual grasp of a single individual.



### Democratic Team

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

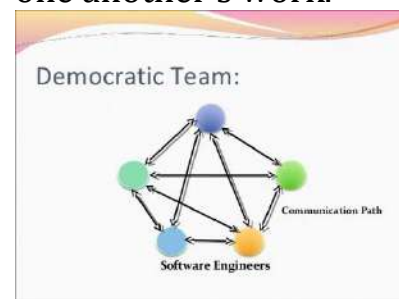
Typically a manager provides the administrative leadership. At different times, different members of the group provide technical leadership

The democratic organization leads to higher morale and job satisfaction.

The democratic team structure is appropriate for less understood problems, since a group of engineers can invent better solutions than a single individual as in a chief programmer team. A democratic team structure is suitable for projects

requiring less than five or six engineers and for research-oriented projects.

The democratic team organization encourages egoless programming as programmers can share and review one another's work.



### **Mixed Control Team Organization**

The mixed team organization draws upon the from both the democratic organization and chief programmer organization. This team organization incorporates both hierarchical and democratic

set-up. The mixed control team organization suitable for large team sizes. The democratic arrangement at the senior engineers level is decompose the problem into small parts. Each

attempts to find solution to a single part.

This team attempts to find solution to a single part. This team structure is extrem popular and is being used inmany software development companies.

### **STAFFING**

### **CHARACTERISTICS OF A GOOD SOFTWARE ENGINEER**

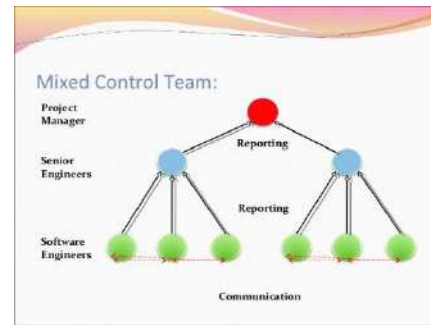
The attributes that smart package engineers ought to posses are as follows:

- Exposure to systematic techniques, i.e., familiarity with package engineer principles.
- Smart technical data of the project areas (Domain knowledge).
- Smart programming talents.
- Smart communication skills. These skills comprise of oral, written, interpersonal skills.
- High motivation.
- Sound data of fundamentals of applied science.
- Intelligence.
- Ability to figure in a very team.
- Discipline, etc.

### **Importance of Risk Identification, Risk Assessment and Risk containment w reference to Risk Management**

Risk management is an emerging area that aims to address the problem identifying and managing the risk associated with a software project. It is really go to identify it, its probability of incident, estimate its impact, and establish emergency plan should the problem actually occur.

The basic motivation of having risk management is to avoid heavy looses. Risk defined as an exposure to the chance of injury or loss. That is risk implies that th is possibility that something negative may happen. In the content of software proj negative implies that there is an adverse effect on cost, quantity or schedule. F



ideas  
the  
report  
is  
used

management aims at reducing the impact of all kinds of risk that might affect the project.

Risk management consists of three essential activities:

- Risk identification
- Risk assessment
- Risk containment

### **Risk Identification**

A project can get affected by a large variety of risks. Risk identification identifies the different risks for a particular project. In order to identify the important risks which might affect a project, it is necessary to categorize risks into different classes. There are three main categories of risks which can affect a software project are:

- **Project Risks**

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

- **Technical Risks**

Technical risks concern potential design, implementation, interfacing, testing, and maintenance problems. Technical risks also include incomplete specifications, changing specifications, technical uncertainty. Most technical risks occur due to development teams' insufficient knowledge about the product.

- **Business risks**

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

### **Risks Assessment**

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

a> The likelihood of a risk coming true (r)

b> The consequence of the problem associated with that risk(s)

The priority of each risk can be computed as

$$p = r * s$$

Where p is the priority with which the risk must be handled, r is the probability of the risk becoming true and s is the severity of damage caused due to the risk becoming true.

### **Risk Containment**

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

- Avoid the risk
- Risk reduction
- Transfer the risk

### **Avoid the Risk**

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

### **Transfer the Risk**

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

### **Risk Reduction**

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

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

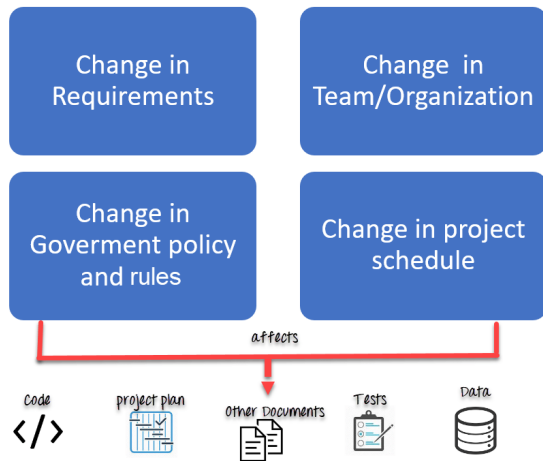
### **SOFTWARE CONFIGURATION MANAGEMENT**

Configuration Management helps organizations to systematically manage, organize and control the changes in the documents, codes, and other entities during Software Development Life Cycle. It is abbreviated as the SCM process. It aims to control cost and work effort involved in making changes to the software system. Its primary goal is to increase productivity with minimal mistakes.

### **Why do we need Configuration management?**

The primary reasons for Implementing Software Configuration Management Systems are:

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



## CONFIGURATION MANAGEMENT ACTIVITIES

### Configuration Identification:

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

### Activities during this process:

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

Example:

Instead of naming a File login.php it should be named login\_v1.2.php where v stands for the version number of the file

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

### **Baseline:**

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

### **Activities during this process:**

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

In simple words, baseline means ready for release.

### **Change Control:**

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

### **Activities during this process:**

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

### **Configuration Status Accounting:**

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

### **Activities during this process:**

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



- Complete listing of all changes since the last baseline
- Allows tracking of progress to next baseline
- Allows to check previous releases/versions to be extracted for testing

### **Configuration Audits and Reviews:**

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

### **Activities during this process:**

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

### **UNIT-III**

#### **REQUIREMENTS ANALYSIS AND SPECIFICATION**

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

- Requirements gathering and analysis.
- Requirements specification

A few members of the development team usually visit the customer site for requirement gathering and analysis activity.

System analyst collect data pertaining to the product to be developed and analyze these data to conceptualized what exactly needs to be done. Once the system analyst understands the precise user requirements, he analyse the requirement to weed out inconsistencies, anomalies, and incompleteness. then he proceeds to write a specification document.

The SRS is first reviewed by the project team to ensure that it is understandable, consistent, unambiguous and complete. the SRS document is then given to the customer for review. After the customer has reviewed the SRS and approved it then it forms the basis for all future development activities and also serve as a contract between the customer and development.

#### **REQUIREMENTS GATHERING AND ANALYSIS**

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

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

- Requirements Gathering: The activity involves interviewing the end users and customers and studying the existing documents to collect all possible information regarding the System.
- Analysis of Gathered Requirements : The main purpose of this activity is to clearly understand the exact requirements of the customer

The following basic questions pertaining to the project should be clearly understood by the analyst in order to obtain a good grasp of the problem:

- What is the problem?
- Why is it important to solve the problem?
- What are the possible solutions to the problem?
- What exactly are the data input to the system and what exactly are the



data output by the system?

- What are the likely complexities that might arise while solving the problem?
- If there are external software or hardware with which the developed software has to interface, then what exactly would the data interchange formats with the external system be?

After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various requirements problems. The most important requirements problems that the analyst has to identify and eliminate are problems of

- anomalies,
- inconsistencies,
- incompleteness.

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

**Inconsistency**: Two requirements are said to be inconsistent, if one of the requirements contradicts the other. Two end-users of the system give inconsistent descriptions of the requirement.

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

### **SOFTWARE REQUIREMENT SPECIFICATION**

After the analyst has collected all the required information regarding the software to be developed and has removed all incompleteness, inconsistencies and anomalies from the specification, the 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. Different people need the SRS document for various 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 is accurately described in the SRS document.

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

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 use the software and validate its working.

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

❖ **Project Managers**

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

❖ **Maintenance Engineers**

The SRS document helps the maintenance engineers to understand functionalities of the system. A clear knowledge of the functionalities help them to understand the design and code.

### **CONTENTS OF THE SRS DOCUMENT**

An SRS document should clearly document:

❖ **Functional Requirements**

❖ **Non functional Requirements**

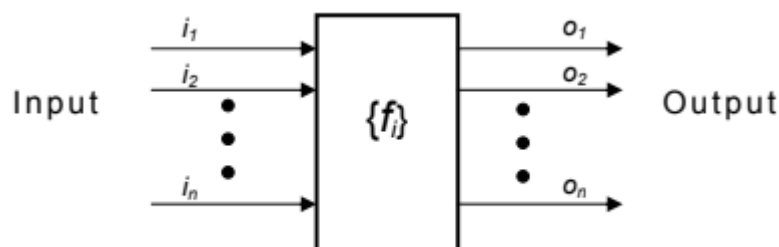
❖ **Goals of implementation**

#### **Functional Requirement**

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

The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high-level functions  $\{f_i\}$ .

Each function  $f_i$  of the system can be considered as a transformation of a set of input data  $(i_i)$  to the corresponding set of output data  $(o_i)$ . The user can get some meaningful piece of work done using a high-level function.



#### **NONFUNCTIONAL REQUIREMENTS:-**

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

Non-functional requirements may include:

# reliability issues,

- # accuracy of results,
- # human - computer interface issues,
- # constraints on the system implementation, etc.

### **GOALS OF IMPLEMENTATION:-**

The goals of implementation part documents some general suggestions regarding development. These suggestions guide trade-off among design goals. The goals implementation section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in future, reusability issues, etc. These are the items which the developers might keep their mind during development so that the developed system may meet some aspects that are not required immediately.

The high-level functional requirements often need to be identified either from informal problem description document or from a conceptual understanding of problem. Each high-level requirement characterizes a way of system usage by some user to perform some meaningful piece of work. There can be many types of users of a system and their requirements from the system may be very different. So, it is useful to identify the different types of users who might use the system and then to identify the requirements from each user's perspective.

Here we list all functions  $\{f_i\}$  that the system performs. Each function  $f_i$  as shown in fig. is considered as a transformation of a set of input data to some corresponding output data.



Example:-

Consider the case of the library system, where -

F1: Search Book function (fig. 3.3)

Input: an author's name

Output: details of the author's books and the location of these books in the

## **DOCUMENTING FUNCTIONAL REQUIREMENTS**

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

. Let us first try to document the withdraw-cash function of an ATM (Automated Teller Machine) system. The withdraw-cash is a high-level requirement. It has several sub-requirements corresponding to the different user interactions. These different interaction sequences capture the different scenarios.

Example: - Withdraw Cash from ATM

R1: withdraw cash

Description: The withdraw cash function first determines the type of account that

the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash, otherwise it generates an error message.

R1.1 select withdraw amount option

Input: "withdraw amount" option

Output: user prompted to enter the account type

R1.2: select account type

Input: user option

Output: prompt to enter amount

R1.3: get required amount

Input: amount to be withdrawn in integer values greater than 100 and less than

10,000 in multiples of 100.

Output: The requested cash and printed transaction statement.

Processing: the amount is debited from the user's account if sufficient balance is

available, otherwise an error message displayed.

### **PROPERTIES OF A GOOD SRS DOCUMENT**

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

**CONCISE.** The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability; also increase error possibilities.

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

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

**CONCEPTUAL INTEGRITY.** It should show conceptual integrity so that the user can easily understand it.

**RESPONSE TO UNDESIRED EVENTS.** It should characterize acceptable responses to undesired events. These are called system responses to exceptional conditions.

**VERIFIABLE.** All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to determine whether or not requirements have been met in an implementation.

### **ORGANIZATION OF THE SRS DOCUMENT**

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

1. Introduction

(a) Background

(b) Overall Description

(c) Environmental Characteristics

(i)Hardware  
(ii)Peripherals  
(iii)People  
Goals of implementation  
Functional requirements  
Non-functional Requirements  
Behavioural Description  
(a) System States  
(b)Events and Actions

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

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

### **The environmental characteristics.**

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

- For example-the hardware that the system will run on.
- The devices that the system will interact
- The user skill level

### **Goals of implementation**

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

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

### **Nonfunctional Requirements**

- Reliability                      Maintainability      Portability

### **Behavioral Description**

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



## **TECHNIQUES FOR REPRESENTING COMPLEX LOGIC:-**

Good SRS documents sometimes may have the conditions which are complex which may have overlapping interactions & processing sequences. There are 1 main techniques available to analyze & represent complex processes logic are.

A) Decision tree

B) Decision table

After these two techniques are finalizing the decision making logic is captured in form of tree or table. We can automatically find out the test cases to validate decision from the decision making logic. These two techniques are widely used in applications of information theory & switching theory.

### **DECISION TREE**

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

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

- **New member**
- **Renewal**
- **Cancel membership**

#### **New member option**

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

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

#### **Renewal option**

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

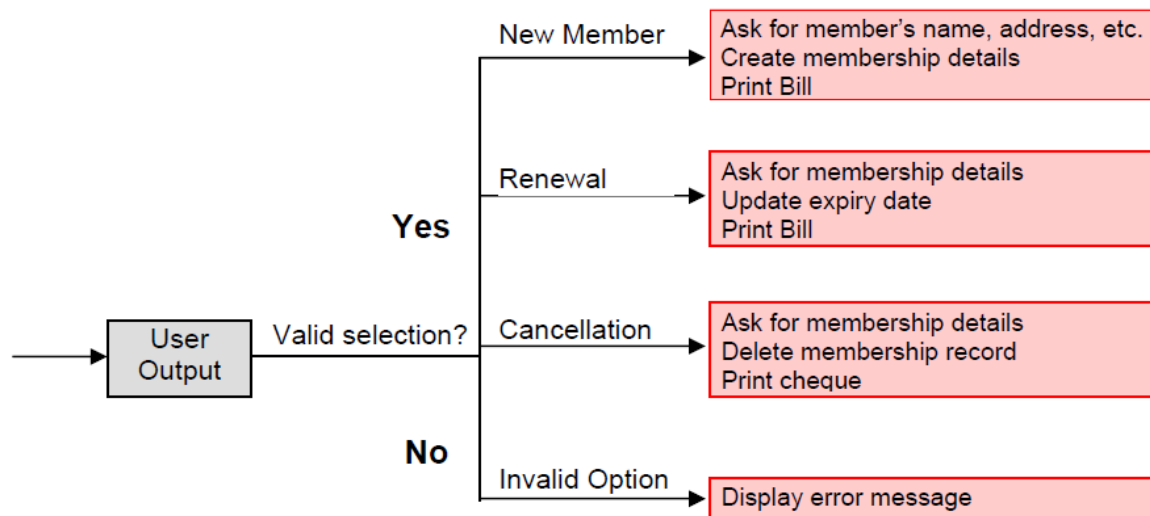
**Action:** If the membership is valid then membership expiry date is updated and annual membership bill is printed, otherwise an error message is displayed.

#### **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 member is printed and finally the membership record is deleted from the database.

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



### DECISION TABLE

A decision table is used to represent the complex processing logic in a tabular or 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 corresponding conditions are satisfied. A column in a table is called a rule. A rule implies that if a condition is true, then the corresponding action is to be executed.

Example: - Consider the previously discussed LMS example. The following decision table shows how to represent the LMS problem in a tabular form. Here the table is divided into two parts, the upper part shows the conditions and the lower part shows what actions are taken. Each column of the table is a rule.



**Conditions**

Valid selection	No	Yes	Yes	Yes
New member	-	Yes	No	No
Renewal	-	No	Yes	No
Cancellation	-	No	No	Yes

**Actions**

Display error message	x	-	-	-
Ask member's details	-	x	-	-
Build customer record	-	x	-	-
Generate bill	-	x	x	-
Ask member's name & membership number	-	-	x	x
Update expiry date	-	-	x	-
Print cheque	-	-	-	x
Delete record	-	-	-	x

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

## UNIT-IV

### SOFTWARE DESIGN

Software design and its activities Software design deals with transforming customer requirements, as described in the SRS document, into a form (a set of documents) that is suitable for implementation in a programming language. A good software design is seldom arrived by using a single step procedure but rather through several iterations through a series of steps. Design activities can be broadly classified into two important parts:

- Preliminary (or high-level) design and
- Detailed design.

Preliminary and detailed design activities

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

**High-level design** means identification of different modules and the relationships among them and the definition of the interfaces among these modules. The outcome of high-level design is called the program structure or software architecture. Many different types of notations have been used to represent a high-level design. A popular way is to use a tree-like diagram called the structure chart to represent the control hierarchy in a high-level design.

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

Items developed during the software design phase For a design to be easily implemented in a conventional programming language, the following items must be designed during the design phase.

- Different modules required to implement the design solution.
- Control relationship among the identified modules. The relationship is also known as the call relationship or invocation relationship among modules.
- 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 each individual module.

### **CHARACTERISTICS OF A GOOD SOFTWARE DESIGN**

The definition of “a good software design” can vary depending on the application being designed. For example, the memory size used by a program may be an important issue to characterize a good solution for embedded software development – since embedded applications are often required to be implemented using memory of limited size due to cost, space, or power consumption considerations. In embedded applications, one may sacrifice design comprehensibility to achieve compactness. For general applications, factors like design comprehensibility may take a back seat while judging the goodness of design. Therefore, the criteria used to judge how good a given design solution is can vary widely depending upon the application. Not only is the goodness of design dependent on the target application, but also the notion of goodness of a design itself varies widely across software engineers and academicians. However, most researchers and software engineers agree on a few desirable characteristics that every good software design for general application must possess. The characteristics are listed below:

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

Possibly the most important goodness criterion is design correctness. A design has to be correct to be acceptable. Given that a design solution is correct, understandability of a design is possibly the most important issue to be considered while judging the goodness of a design. A design that is easy to understand is also easy to develop, maintain and change. Thus, unless a design is easily understandable, it would require tremendous effort to implement and maintain it.

**Features of a design document** In order to facilitate understandability, the design document should have the following features:

- It should use consistent and meaningful names for various design components.

- The design should be modular. The term modularity means that it should use a cleanly decomposed set of modules.
- It should neatly arrange the modules in a hierarchy, e.g. in a tree-like diagram

### **Modularity**

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



## **CLASSIFICATION OF COHESION**

**COINCIDENTAL COHESION:** A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, the module contains a random collection of functions. It is likely that the functions have been put in the module out of pure coincidence without any thought or design. For example, in a transaction processing system (TPS), the 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. An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.

**TEMPORAL COHESION:** When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion. 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 are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective, e.g. the algorithm for decoding a message.

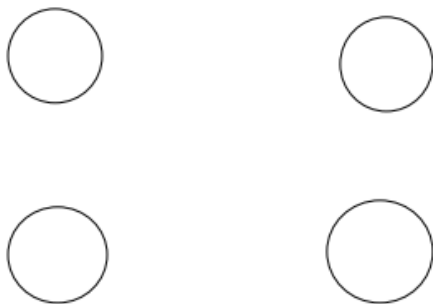
**COMMUNICATIONAL COHESION:** A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, e.g. the set of functions defined on an array or a stack. **Sequential cohesion:** A module is said to possess sequential cohesion, if the elements of a module form the part of a sequence, where the output from one element of the sequence is input to the next. For example, in a TPS, the get-input, validate-input, sort-input functions are grouped into one module.

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

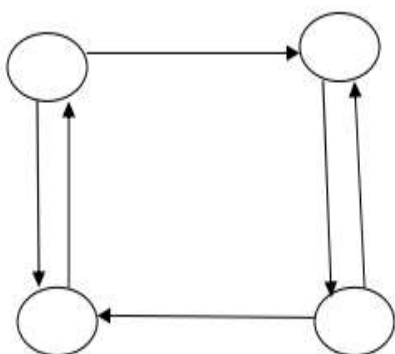
cohesion. Suppose a module exhibits functional cohesion and we are asked describe what the module does, then we would be able to describe it using a single sentence.

### **COUPLING**

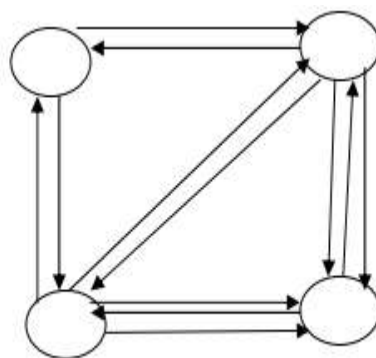
Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. A module having high cohesion and low coupling is said to be functionally independent of other modules. If two modules interchange large amounts of data, then they are highly interdependent. The degree of coupling between two modules depends on their interface complexity. Interface complexity is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.



Uncoupled: No Dependencies



Loosely coupled: some dependencies



Highly coupled: many dependencies

Classification of Coupling Even if there are no techniques to precisely quantitatively estimate the coupling between two modules, classification of different types of coupling will help to quantitatively estimate the degree of coupling.



between two modules. Five types of coupling can occur between any two modules. This is shown in fig.

. Data Stamp Control Common Content

Low High

## **CLASSIFICATION OF COUPLING**

**Data coupling:** Two modules are data coupled, if they communicate through a parameter. An example is an elementary data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for the control purpose.

**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. An example of control coupling is 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

Object-oriented design

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

- assign-membership-number
- create-member-record
- print-bill Each of these sub functions may be split into more detailed sub functions and so on.

**Object Oriented Design**

In the object-oriented design approach, the system is viewed as a collection of objects. The system state is decentralized among the objects and each object manages its own state information. 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. Structured analysis is used to carry out the top-down decomposition of the set of high-level functions depicted in the problem description and to represent them graphically. 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).


### **USE OF DATA FLOW DIAGRAM**


The DFD also known as bubble chart is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on these data & the output data generated by the system. It is a very simple formalism – it is simple to understand and use. A DFD model uses a very limited number of primitive symbols to represent the functions performed by the system and the dataflow among these functions.


### **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 using a circle.

External entity symbol (  ) : An external entity is 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  $\{ \equiv \}$  : A data store represents a logical file. It is represented using two parallel lines.

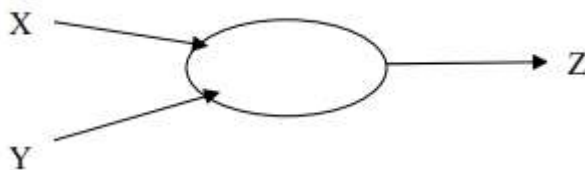
Output symbol  $\square$  : The output symbol is used when a hard copy is produced or the user of the copies cannot be clearly specified or there are several users of output.

### **CONSTRUCTION OF DFD**

A DFD model of a system graphically represents how each input data is transformed to its corresponding output data through a hierarchy of DFDs. A DFD starts 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.



### **LEVEL 1 DFD**

The level 1 DFD usually contains between 3 and 7 bubbles. To develop the Level 1 DFD, examine the high-level functional requirements. If there are between 3 to 7 high-level functional requirements, then these can be directly represented as bubbles in the Level 1 DFD. We can examine the input data to these functions and the data output by these functions and represent them appropriately in the diagram. 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 one bubble in the Level 1 DFD.

### **DECOMPOSITION**

Each bubble in the DFD represents a function performed by the system. ' bubbles are decomposed into sub functions at the successive level of the DFD. Each bubble at any level of DFD is usually decomposed between three to seven bubbles. Decomposition of a bubble should be carried out until a level is reached at

### DFD for Library Management System

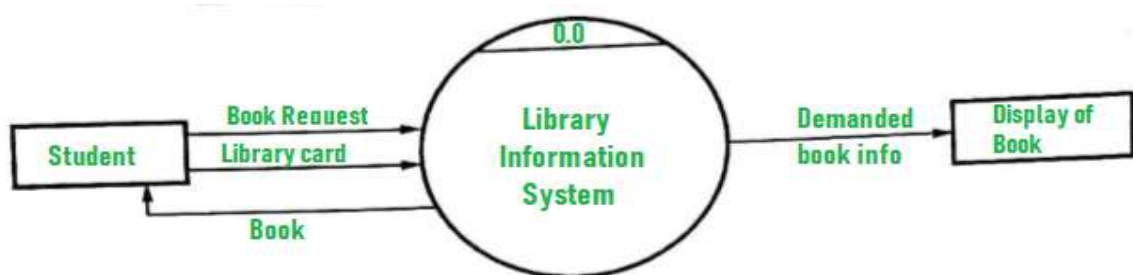
Data Flow Diagram (DFD) depicts the flow of information and the transformation applied when a data moves in and out from a system. The overall system is represented and described using input, processing and output in the DFD. The inputs can be:

- **Book request** when a student requests for a book.
- **Library card** when the student has to show or submit his/her identity as a proof.

The overall processing unit will contain the following outputs that a system can produce or generate:

- Book will be the output as the book demanded by the student will be given to them.
- Information of demanded book should be displayed by the library information system that can be used by the student while selecting a book which makes it easier for the student.

#### Level 0 DFD -



#### Level 1 DFD

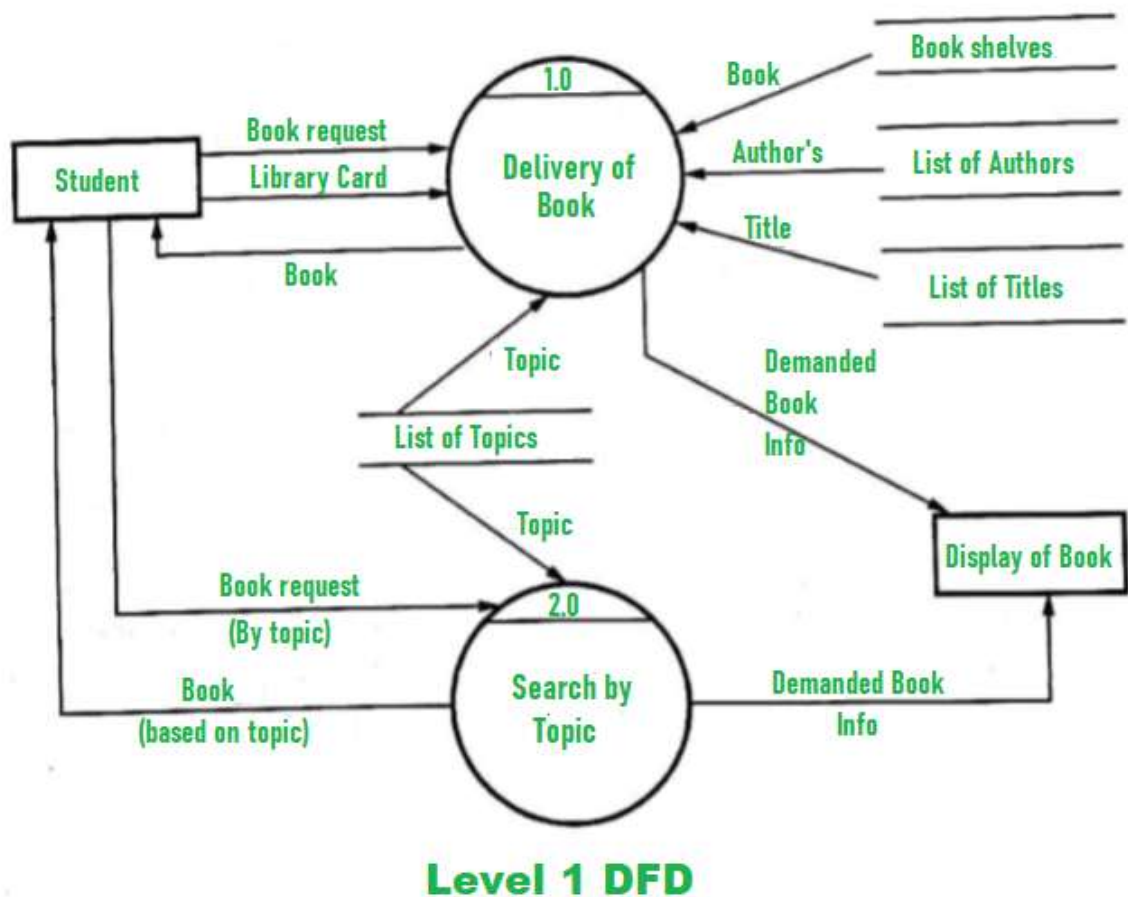
At this level, the system has to show or exposed with more details of process. The processes that are important to be carried out are:

Book delivery

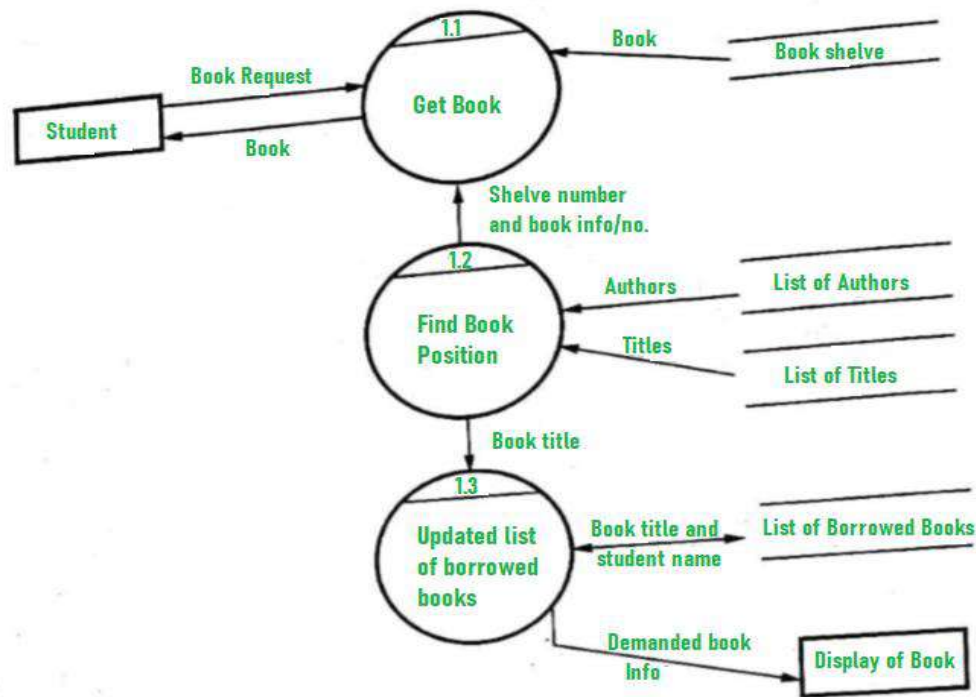
Search by topic

List of authors, List of Titles, List of Topics, the bookshelves from which books can be located are some information that is required for these processes. Data store is used to

represent this type of information.



Level 2 DFD –



**Level 2 DFD**

### LIMITATIONS OF DFD

- ◆ A data flow diagram does not show flow of control. It does not show details linking inputs and outputs within a transformation. It only shows all possible inputs and outputs for each transformation in the system.
- ◆ The method of carrying out decomposition to arrive at the successive levels up to the ultimate level to which decomposition is carried out are highly subjective and 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 subfunctions.
- ◆ 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 (i.e. a DFD representation) into a structure chart. Structured design provides 1 strategies to guide transformation of a DFD into a structure chart.

- Transform analysis
- 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 lower-level DFDs. At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD.

## **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 module) and the parameters that are passed among the different modules. Hence, structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on the module structure of the software and the interactions among different modules, procedural aspects (e.g. 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.

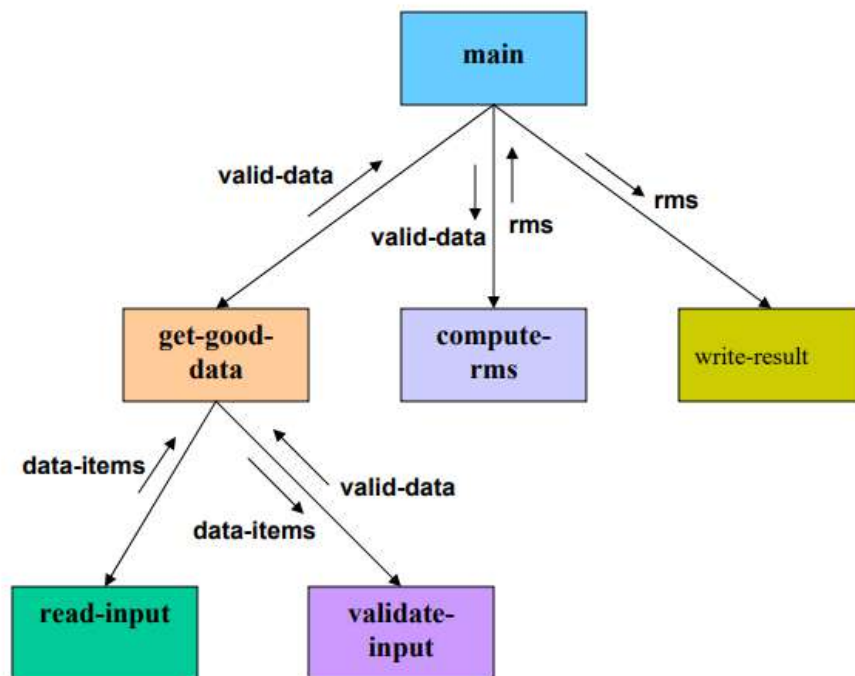
## **TRANSFORM ANALYSIS**

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

- Input
- Logical processing
- Output

The input portion in the DFD includes processes that transform input data 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 the

central transform. In the next step of transform analysis, the structure chart derived by drawing one functional component for the central transform and afferent and efferent branches. Identifying the highest level input and output transforms require experience and skill. The first level of structure chart is produced by representing each input and output unit as boxes and each central transform as a single box. In the third step of transform analysis, the structure chart is refined by adding subfunctions required by each of the high-level functional components. Multiple 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, error-handling modules, initialization and termination procedures, etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.



### Transaction Analysis

A transaction allows the user to perform some meaningful piece of work. In a transaction-driven system, one of several possible paths through the DFD is traversed depending upon the input data item. Each different way in which input data is handled in a transaction defines a transaction. The number of bubbles on which the input data item is incident defines the number of transactions. Some transactions may require any input data. For each identified transaction, we trace the input data to output. In the structure chart, we draw a root module and below this module draw each identified transaction of a module.



## UNIT-V

### USER INTERFACE DESIGN

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

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

- f Use of Metaphors and intuitive command names.

Speed of learning an interface is greatly facilitated if these are based on some day-to-day real-life examples or some physical objects with which the users are familiar. The abstractions of real-life objects or concepts used in user interface design are called metaphors. If the user interface of a text editor uses concepts similar to the tools used by a writer for text editing such as cutting lines and paragraphs and pasting it at other places, users can immediately relate to it. Another popular metaphor is a shopping cart. Everyone knows how a shopping cart is used to make choices while purchasing items in a supermarket. If a user interface uses a shopping cart metaphor for designing the interaction style for a situation where similar types of choices have to be made, then the users can easily understand and learn to use the interface. Yet another example of a metaphor is the trashcan. To delete a file, the user may drag it to the trashcan. Also, learning is facilitated by intuitive command names and symbolic command issue procedures.

- . f Consistency. Once a user learns about a command, he should be able to use the similar commands in different circumstances for carrying out similar actions. This makes it easier to learn the interface since the user can extend his knowledge about one part of the interface to the other parts. For example, in a word processor, "Control-b" is the short-cut key to embolden the selected text. The same short-cut should be used on the other parts of the interface, for example, to embolden text, graphic objects also - circle, rectangle, polygon, etc. Thus, the different commands supported by an interface should be consistent.

- f Component-based interface. Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications.

with which the user is already familiar. This can be achieved if the interfaces of different applications are developed using some standard user interface components. This, in fact, is the theme of the component-based user interface. Examples of standard user interface components are: radio button, check box, text field, slider, progress bar, etc. The speed of learning characteristic of a user interface can be determined by measuring the training time and practice that users require before they can effectively use the software.

**Speed of use.** Speed of use of a user interface is determined by the time and user effort necessary to initiate and execute different commands. This characteristic of the interface is sometimes referred to as productivity support of the interface. 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 can be achieved through careful design of the interface. For example, an interface that requires users to type in lengthy commands or involves mouse movements to different areas of the screen that are wide apart for issuing commands can slow down the operating speed of users. The most frequently used commands should have the smallest length or be available at the top of the menu to minimize mouse movements necessary to issue commands.

which can record the frequency and types of user error and later display statistics of various kinds of errors committed by different users. Moreover, errors can be prevented by asking the users to confirm any potentially destructive actions specified by them, for example, deleting a group of files. Consistency of names, icons, procedures, and behavior of similar commands and the simplicity of the command issue procedures minimize error possibilities. Also, the interface should prevent user from entering wrong values.

- **Attractiveness.** A good user interface should be attractive to use. An attractive user interface catches user attention and fancy. In this respect, graphical 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 knowledge about aspects of the interface from one part to another. Thus, consistency facilitates speed of learning, speed of recall, and also helps in reduction of error rate.

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



periodically informed about the progress made in processing his command. For example, if the user specifies a file copy/file download operation, a progress bar should be displayed to display the status. This will help the user to monitor the status of action initiated.

- Support for multiple skill levels. A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users. This is necessary because users with different levels of experience in using an application prefer different types of user interfaces. Experienced users are more concerned about the efficiency of the command issue procedure, whereas novice users pay importance to usability aspects. Very cryptic and complex commands discourage a novice, whereas elaborate command sequences make the command issue procedure very slow and therefore put off experienced users. When someone uses an application for the first time, his primary concern is speed of learning. As he uses an application for extended periods of time,

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

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

As the user becomes familiar with the operation of the software. As a user becomes more and more familiar with an interface, his focus shifts from usability aspects to speed of command issue aspects. Experienced users look for options such as “hot keys”, “macros”, etc. Thus, the skill level of users improves as they keep using the software product and they look for commands to suit their skill levels.

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

- 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.

User guidance and online help Users may seek help about the operation of software any time while using the software. This is provided by the on-line help system. This is different from the guidance and error messages which are flashed automatically without the user asking for them. The guidance messages prompt the user regarding the options he has regarding the next command, and the status of the last command, etc. On-line Help System. Users expect the on-line help messages to be tailored to the context in which they invoke the “help system”. Therefore, a good on-line help system should keep track of what a user is doing while invoking the help system and provide the output message in a context-dependent way. Also, the help messages should be tailored to the user’s experience level. Further, a good on-line help system should take advantage of any graphics and animation characteristics of the screen and should not just be a copy of the user’s manual. Fig. 9.1 gives a snapshot of a typical on-line help provided by a user interface.



**Fig. 9.1. Example of an on-line help interface**

Guidance Messages. The guidance messages should be carefully designed to prompt the user about the next actions he might pursue, the current status of the system, the progress made so far in processing his last command, etc. A good guidance system should have different levels of sophistication for different categories of users. For example, a user using a command language interface might need a different type of guidance compared to a user using a menu or icon interface. Also, users should have an option to turn off detailed messages.

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

determined by the sequence of commands already issued by the user. A mode-based interface can be represented using a state transition diagram, where each node of state transition diagram would represent a mode. Each state of the state transition diagram can be annotated with the commands that are meaningful in that state.

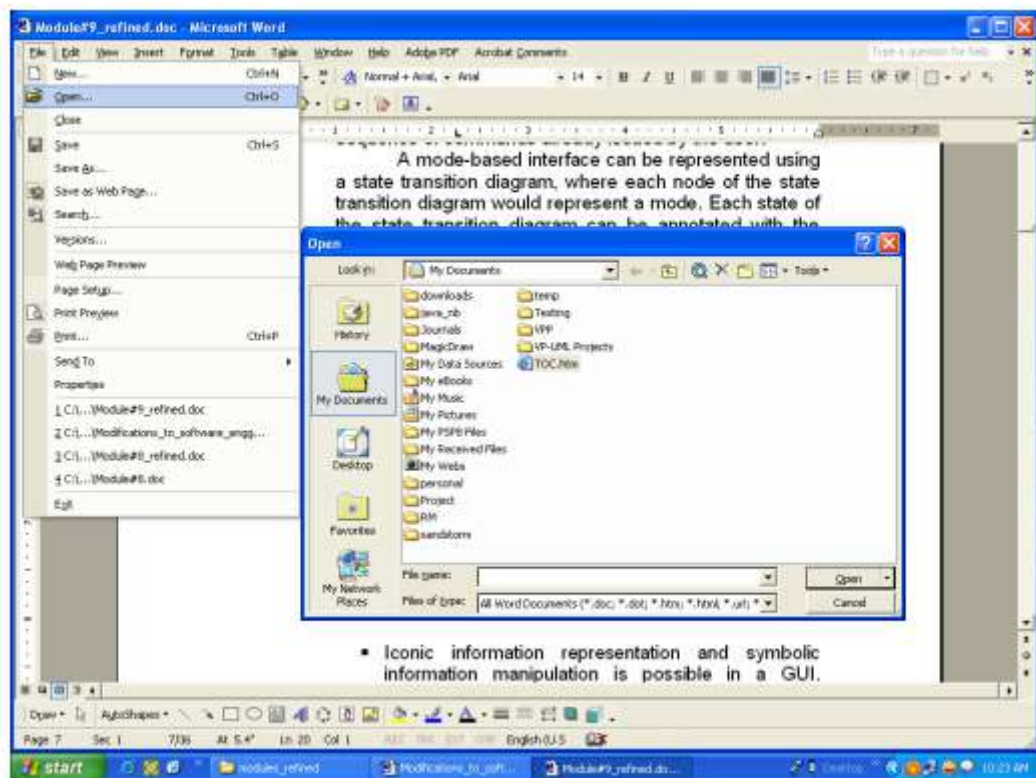


Fig 9.2 shows the interface of a word processing program. The top-level menu provides the user with a gamut of operations like file open, close, save, etc. When user chooses the open option, another frame is popped up which limits the user to select a name from one of the folders.

**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 a 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 are 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 : the user can instantly remember it.

- A GUI usually supports command selection using an attractive and user friendly menu selection system. f 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.
- On the flip side, a GUI requires special terminals with graphics capabilities running and also requires special input devices such a mouse. On the other hand, a text-based user interface can be implemented even on a cheap alphanumeric display terminal. Graphics terminals are usually much more expensive than alphanumeric terminals. However, display terminals with graphics capability with bit-mapped high-resolution displays and significant amount of local processing power have become affordable and over the years have replaced text-based terminals on all desktops. Therefore, the emphasis in this lesson is on GUI design rather than textbased user interface design.

### **TYPES OF USER INTERFACE**

User interfaces broadly classified into three categories:

Command language-based interfaces

Menu-based interfaces Direct

Direct manipulation interfaces

### **COMMAND LANGUAGE-BASED INTERFACES**

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

require the user to memorize the set of primitive commands. Most users make errors while formulating commands in the command language and also while typing them in. In a command language-based interface, all interactions with the system are through a keyboard and cannot take advantage of mouse. For inexperienced users, command language-based interfaces are not suitable.

## **ISSUES IN DESIGNING A COMMAND LANGUAGE INTERFACE**

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

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

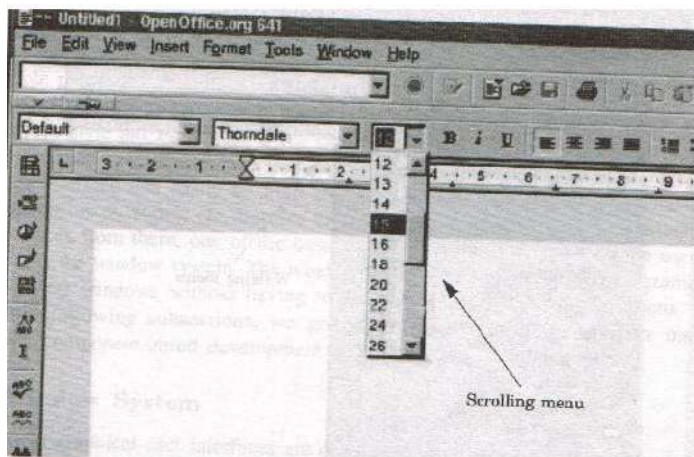
The designer has to decide whether it should be possible to compose primitive commands to form more complex commands. A sophisticated command composition facility would require the syntax and semantics of the various command composition options to be clearly and unambiguously specified. The ability to combine commands can be usefully exploited by experienced users, but is quite unnecessary for inexperienced users.

## **MENU-BASED INTERFACES**

The advantage of a menu-based interface over a command language-based interface is that menu-based interface does not require the users to remember the exact syntax of the commands. A menu-based interface is based on recognition of command names. In this type of interface the typing effort is minimal as most interactions are carried out through menu selections using a pointing device. Experienced users find a menu-based user interface to be slower than a command language-based interface because they can type fast and get speed advantage in composing different primitive commands to express complex commands. Composing commands in a menu-based interface is not possible. A major challenge in the design of a menu-based interface is that of structuring the large number of menu choices into manageable forms. The techniques available to structure menu items are:

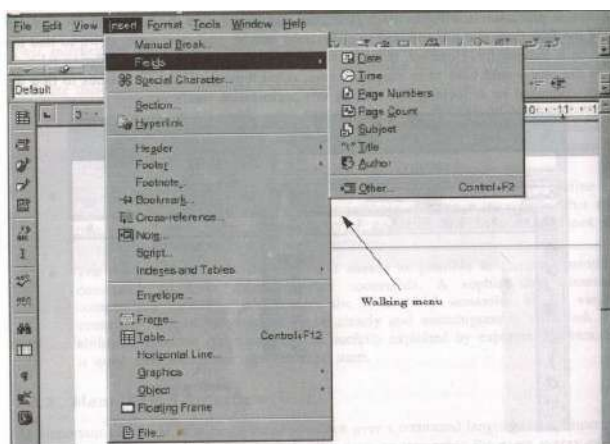
**SCROLLING MENU** When a full choice list cannot be displayed within the menu area, scrolling of the menu items is required. This enables the user to view and select menu items that cannot be accommodated on the screen.





## WALKING MENU

Walking menu is a very commonly used menu to structure a large collection of menu items. In this technique, when a menu item is selected, it causes further menu items to be displayed adjacent to it in a sub-menu. A walking menu can be successfully used to structure commands only if there are limited choices since each adjacent displayed menu does take up screen space and the total screen area, after all, is limited.



## HIERARCHICAL MENU:

In this technique, the menu items are organized in a hierarchy or tree structure. Selecting a menu item causes the current menu display to be replaced by an appropriate sub-menu. Walking menu can be considered to be a form of hierarchical menu. Hierarchical menu, on the other hand, can be used to manage a large number of choices, but the users are likely to face navigational problems and therefore lose track of their whereabouts in the menu tree. This probably is the main reason why this type of interface is very rarely used.

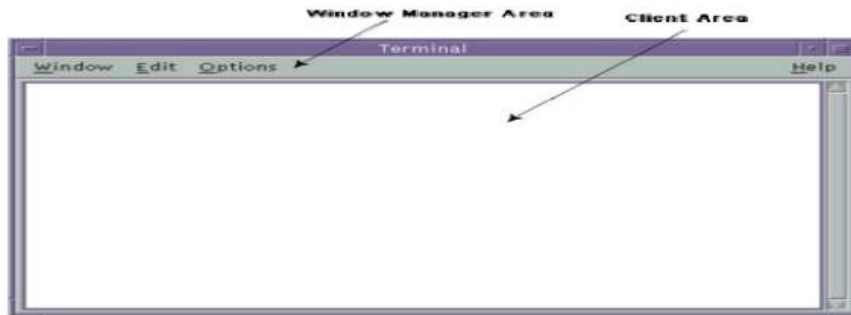
**DIRECT MANIPULATION INTERFACES** Direct manipulation interfaces present interface to the user in the form of visual models i.e. icons. This type of interface is called as iconic interface. In this type of interface, the user issues commands performing actions on the visual representations of the objects.

The advantages of iconic interfaces are that the icons can be recognised by the user very easily and icons are language-independent.

**Main aspects of Graphical UI, Text based Interface Aspects of GUI** In a GUI, multiple windows with different information can simultaneously be displayed on the user's screen. Iconic information representation and symbolic information manipulation are possible in a GUI. Symbolic information manipulation, such as dragging an icon representing a file to a trash can for deleting, is intuitively very appealing and the user can instantly remember it. 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 of the command issue procedure. A GUI flip side, a GUI requires special terminals with graphics capabilities for running and also requires special input devices such as a mouse. **Text Based Interface** Text based interface can use text, symbols and colours available on a given text environment. Text-based user interface can be implemented on a cheap alphanumeric display terminal.

## **WINDOW**

A window is a rectangular area on the screen. A window can be considered to be a virtual screen, in the sense that it provides an interface to the user for carrying out independent activities, e.g. one window can be used for editing a program and another for drawing pictures, etc. A window can be divided into two parts: client part, and non-client part. The client area makes up the whole of the window, except for the borders and scroll bars. The client area is the area available to a client application for display. The non-client part of the window determines the look and feel of the window. The look and feel defines a basic behavior for all windows, such as creating, moving, resizing, and iconifying the windows. A basic window with different parts is shown in fig.

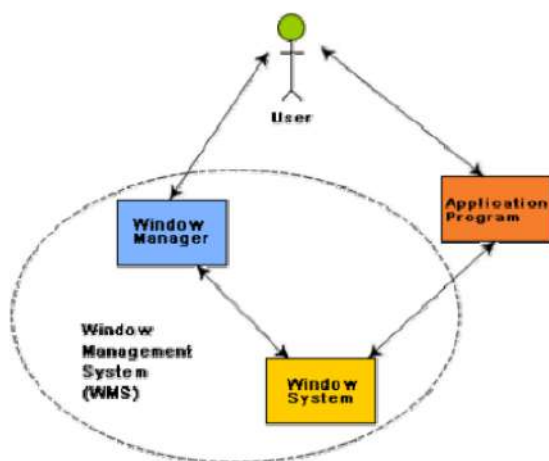


## WINDOW MANAGEMENT SYSTEM

Window Management System (WMS) A graphical user interface typically consists of a large number of windows. Therefore, it is necessary to have some systematic way to manage these windows. Most graphical user interface development environments do this through a window management system (WMS). A window management system is primarily a resource manager. It keeps track of the screen area resources and allocates it to the different windows that seek to use the screen. From a broad perspective, a WMS can be considered as a user interface management system (UIMS) – which not only does resource management, but also provides the basic behavior to the windows and provides several utility routines to the application programmer for user interface development. A WMS simplifies the task of a window designer to a great extent by providing the basic behavior to the various windows such as move, resize, iconify, etc. as soon as they are created and by providing basic routines to manipulate the windows from the application such as creating, destroying, changing different attributes of the windows, and drawing text, lines, etc.

A WMS consists of two parts

- a window manager, and
- a window system.





## **WINDOW MANAGER AND WINDOW SYSTEM**

Window manager is the component of WMS with which the end user interacts to various window-related operations such as window repositioning, window resizing, iconification, etc. The window manager is built on the top of the window system in the sense that it makes use of various services provided by the window system. The window manager and not the window system determines how the windows look and behave. In fact, several kinds of window managers can be developed based on the same window system. The window manager can be considered as a special kind of client that makes use of the services (function calls) supported by the window system. The application programmer can also directly invoke the services of the window system to develop the user interface. The relationship between the window manager, window system, and the application program is shown in fig. 9.7. The figure shows that the end-user can either interact with the application itself or with the window manager (resize, move, etc.) and both the application and the window manager invoke services of the window manager.

### **WINDOW MANAGER**

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

### **TYPES OF WIDGETS**

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

#### **LABEL WIDGET**

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

#### **CONTAINER WIDGET**

These widgets do not stand by themselves, but exist merely to contain other widgets. Other widgets are created as children of the container widget. When a container widget is moved or resized, its children widget also get moved or resized. A container widget has no callback routines associated with it.

#### **POP-UP MENU**

These are transient and task specific. A pop-up menu appears upon pressing mouse button, irrespective of the mouse position.

### **PULL-DOWN MENU**

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

Dialog boxes. We often need to select multiple elements from a selection list. A dialog box remains visible until explicitly dismissed by the user. A dialog box can include areas for entering text as well as values. If an apply command is supported in a dialog box, the newly entered values can be tried without dismissing the box. Through many dialog boxes ask you to enter some information, there are some dialog boxes which are merely informative, alerting you to a problem with your system or an error you have made. Generally, these boxes ask you to read the information presented and then click OK to dismiss the box.

### **PUSH BUTTON**

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

### **RADIO BUTTONS**

A set of radio buttons is used when only one option has to be selected out of many options. A radio button is a hollow circle followed by text describing the option it stands for. When a radio button is selected, it appears filled and the previously selected radio button from the group is unselected. Only one radio button from a group can be selected at any time. This operation is similar to that of the band selection buttons that were available in old radios.

### **COMBO BOXES**

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

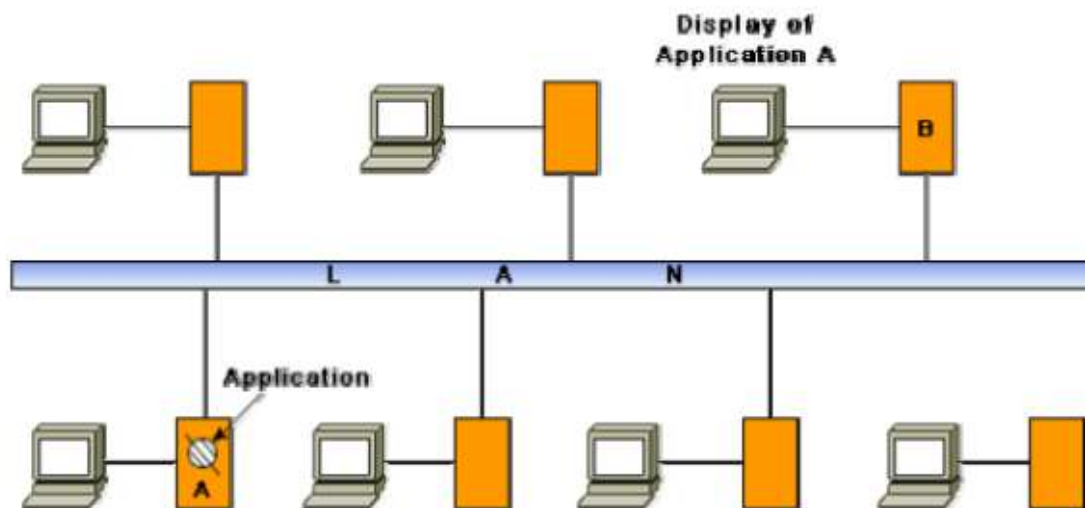
### **X-WINDOW**

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

set of basic widgets and a set of routines to manipulate these widgets. One of the most widely used widget sets is X/Motif. Digital Equipment Corporation (DEC) used the basic X-window functions to develop its own look and feel for interface design called DECWindows.

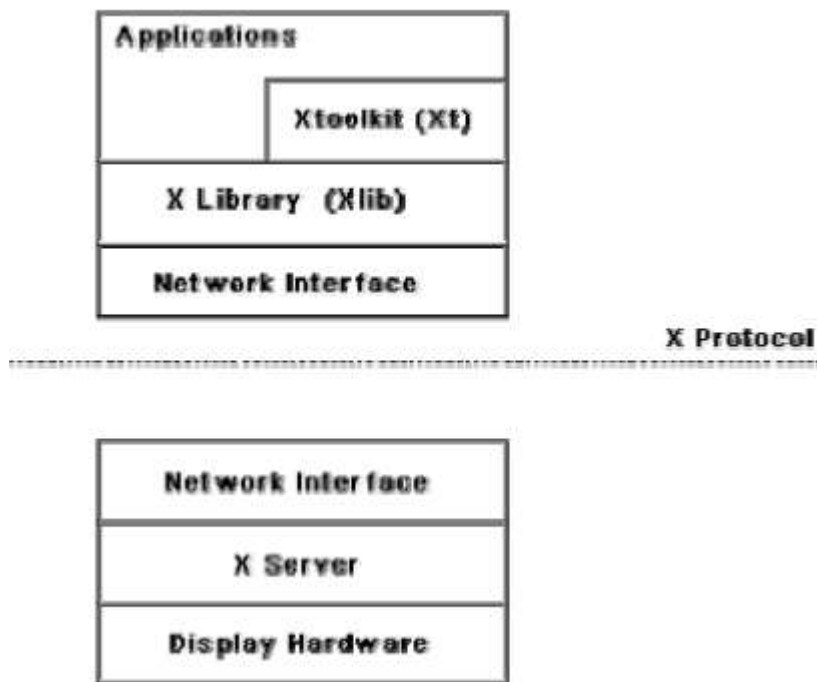
### **POPULARITY OF X-WINDOW**

One of the important reasons behind the extreme popularity of the X-window system is probably due to the fact that it allows development of portable GUIs. Applications developed using the X-window system are device-independent. Also, applications developed using the X-window system become network independent in the sense that the interface would work just as well on a terminal connected anywhere on the same network as the computer running the application is. Network-independent operation has been schematically represented in the fig. 9.8. Here, "A" is the computer application in which the application is running. "B" can be any computer on the network from where interaction with the application can be made. Network-independent GUI was pioneered by the Xwindow system in the mid-eighties at MIT (Massachusetts Institute of Technology) with support from DEC (Digital Equipment Corporation). Nowadays many user interface development systems support network-independent GUI development, e.g. the AWT and Swing components of Java.



### **ARCHITECTURE OF AN X-SYSTEM**

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



## **X-SERVER**

The X server runs on the hardware to which the display and keyboard are attached. The X server performs low-level graphics, manages windows, and user input functions. The X server controls access to a bit-mapped graphics display resource and manages it.

## **X-PROTOCOL**

The X protocol defines the format of the requests between client applications and display servers over the network. The X protocol is designed to be independent of hardware, operating systems, underlying network protocol, and the programming language used.

## **X-LIBRARY (XLIB)**

The Xlib provides a set of about 300 utility routines for applications to call. These routines convert procedure calls into requests that are transmitted to the server. Xlib provides low-level primitives for developing a user interface, such as displaying a window, drawing characteristics and graphics on the window, waiting for specific events, etc.

## **XTOOLKIT (XT)**

The Xtoolkit consists of two parts: the intrinsics and the widgets. We have already seen that widgets are predefined user interface components such as scroll bars, push buttons, etc. for designing GUIs. Intrinsics are a set of about a dozen library routines that allow a programmer to combine a set of widgets into a user interface. In order to develop a user interface, the designer has to put together the set of components (widgets) he needs, and then he needs to define the characteristics (called resources) and behavior of these widgets by using the intrinsic routines to complete the development of the interface. Therefore, developing an interface using Xtoolkit is much easier than developing the same interface using only X library.

### **VISUAL PROGRAMMING**

Visual programming is the drag and drop style of program development. In this style of user interface development, a number of visual objects (icons) representing GUI components are provided by the programming environment. The application programmer can easily develop the user interface by dragging the required component types (e.g. menu, forms, etc.) from the displayed icons and placing them wherever required. Thus, visual programming can be considered as program development through manipulation of several visual objects. Reuse of program components in the form of visual objects is an important aspect of this style of programming. Though popular for user interface development, this style of programming can be used for other applications such as Computer-Aided Design application (e.g. factory design), simulation, etc. User interface development using a visual programming language greatly reduces the effort required to develop a user interface. Examples of popular visual programming languages are Visual Basic, Visual C++, etc. Visual C++ provides tools for building programs with window-based user interfaces for Microsoft Windows environments. In Visual C++, menu bars, icons, and dialog boxes, etc. can be designed easily before adding them to the program. These objects are called as resources. Shape, location, type, and size of the dialog boxes can be designed before writing any C++ code for the application.

## UNIT-VI

### CODING AND TESTING

#### **CODING**

Good software development organizations normally require their programmers adhere to some well-defined and standard style of coding called coding standard. Most software development organizations formulate their own coding standard that suit them most, and require their engineers to follow these standards rigorously. The purpose of requiring all engineers of an organization to adhere to a standard style of coding is the following:

- A coding standard gives a uniform appearance to the codes written by different engineers.
- It enhances code understanding.
- It encourages good programming practices.

A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, error reporting conventions, etc.

#### **CODING STANDARDS AND GUIDELINES**

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

The following are some representative coding standards.

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

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

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



- Different functions supported, along with their input/output parameters.
- Global variables accessed/modified by the module.

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

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

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

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

**Avoid obscure side effects:** The side effects of a function call include modification of parameters passed by reference, modification of global variables, and other operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code. For example, if a global variable is changed obscurely in a called module, some file I/O is performed which is difficult to infer from the function's name and header information, it becomes difficult for anybody trying to understand the code.

**Do not use an identifier for multiple purposes:** Programmers often use the same identifier to denote several temporary entities. For example, some

programmers use a temporary loop variable for computing and a variable for storing the final result. The rationale that is usually given by these programmers for such multiple uses of variables is memory efficiency, e.g. three variables use up three memory locations, whereas the same variable used in three different ways uses just one memory location. However, there are several things wrong with this approach; hence it should be avoided. Some of the problems caused by use of variables for multiple purposes are as follows:

- Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes. Use of a variable

multiple purposes can lead to confusion and make it difficult for someone trying to read and understand the code.

- Use of variables for multiple purposes usually makes future enhancements more difficult.

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

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

### Aim of testing

The aim of the testing process is to identify all defects existing in a software product. However for most practical systems, even after satisfactorily carrying out the test phase, it is not possible to guarantee that the software is error free. This is because of the fact that the input data domain of most software products is very large. It is not practical to test the software exhaustively with respect to each value that the input data may assume. Even with this practical limitation of the testing process, the importance of testing should not be underestimated. It must be remembered that testing does expose many defects existing in a software product. Thus testing provides a practical way of reducing defects in a system and increasing the user confidence in a developed system.

### Program Testing

Testing a program consists of providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected. If the program fails to behave as expected, then the conditions under which failure occurs are noted for later debugging and correction.

Some commonly used terms associated with testing are:

- **Failure:** This is a manifestation of an error (or defect or bug). But, the mere presence of an error may not necessarily lead to a failure.



- Test case: This is the triplet [I,S,O], where I is the data input to the system, S is state of the system at which the data is input, and O is the expected output of system.
- Test suite: This is the set of all test cases with which a given software product is to be tested.

### **Differentiate between verification and validation.**

Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase, whereas validation is the process of determining whether a fully developed system conforms to its requirements specification. Thus while verification is concerned with the containment of errors, the aim of validation is that the final product be error free.

### *CodeWalk-Through*

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

In this technique, after a module has been coded, it is successfully compiled and all syntax errors are eliminated. Some members of the development team are given the code a few days before the walk-through meeting to read and understand the code. Each member selects some test cases and simulates execution of the code through different statements and functions of the code. Even though a code walkthrough is an informal analysis technique, several guidelines have evolved for making this technique more effective and useful. Some guidelines are:

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

## Code Inspection and Software Documentation` CodeInspection

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
- ✓ Comparison of equality of floating point values.

## Software Documentation`

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

- Good documents enhance understandability and maintainability of a software product. They reduce the effort and time required for maintenance.
- Good documents help the users in effectively exploiting the system.
- Good documents help in effectively overcoming the manpower turnover problem. Even when an engineer leaves the organization, the newcomer can build up the required knowledge quickly.
- 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

### 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

### 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.

*Distinguish among Unit Testing, Integration Testing, and System Testing*

A software product is normally tested in the three levels:

- Unit testing
- Integration testing

- Systemtesting

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

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. The integration tests do a more convincing job of demonstrating the system works (especially to non-programmers) than a set of unit tests.

System tests test the entire system. It is set of test carried out by test engineer against the software(system) developed by developer. In system testing the complete system is configured in a controlled environment and test cases are created to simulate the real time scenarios that occurs in a simulated real life test environment. The purpose of system testing is to validate an application and completeness in performing as designed and to test all functions of the system that is required in real life. the most popular approach of system testing is Black Box testing.

## *UnitTesting*

Unit testing or module testing of different units or modules of a system in isolation.

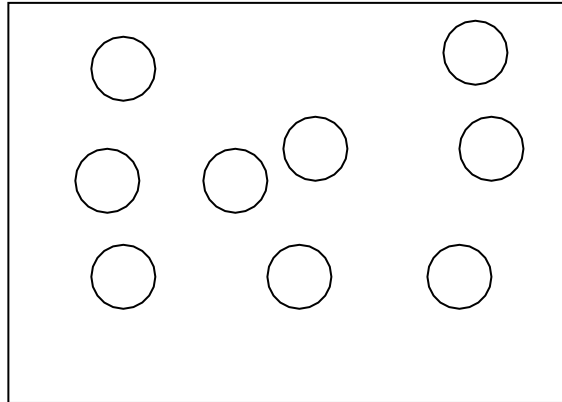


Fig. 6.1 Unit testing

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

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

### *Driver and Stub Modules*

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

- The procedures belonging to other modules that the module under test calls.

- Nonlocal data structures that the module accesses.
- A procedure to call the function of the module under test with appropriate parameters.

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

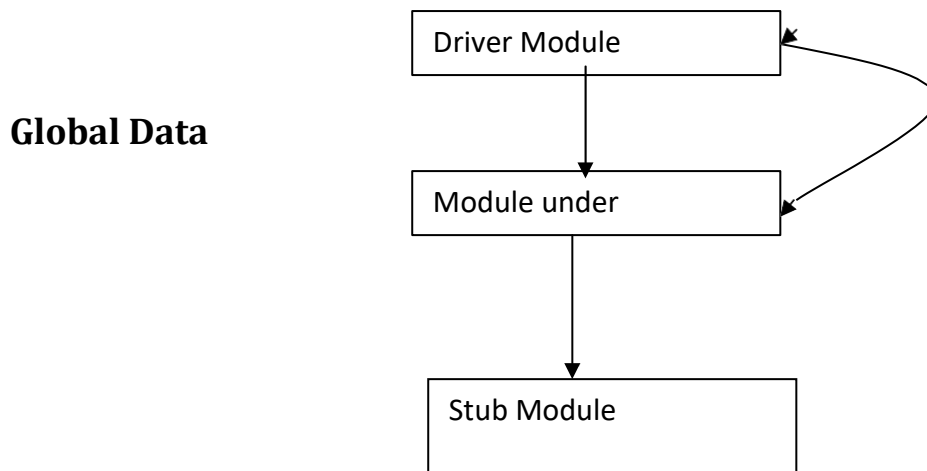


Fig. 6.2 Unit testing with the help of driver and stub module

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

## INTRODUCTION TO BLACK BOX TESTING

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

1. black box approach,
2. white box approach

## **BLACK BOX TESTING**

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



In the black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design, or code is required. The following are the two main approaches to designing black box test cases.

- Equivalence class 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 behaviour of the program is similar for every input data belonging to the same equivalence class. The main idea behind defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that 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}.

Example#2 Let us consider an example of any college admission process. There is college that gives admissions to students based upon their percentage.

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



Example#3: Design the black-box test suite for the following program. The program computes the intersection point of two straight lines and displays the result. It reads two integer pairs (m1, c1) and (m2, c2) defining the two straight lines of the form

$$y=mx + c.$$

The equivalence classes are the following:

- Parallel lines (m1=m2, c1≠c2)
- Intersecting lines (m1≠m2)
- Coincident lines (m1=m2, c1=c2) Now, selecting one representative value from each equivalence class, the test suit (2, 2) (2, 5), (5, 5) (7, 7), (10, 10) (10, 10) are obtained.

### **BOUNDARY VALUE ANALYSIS**

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

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}.

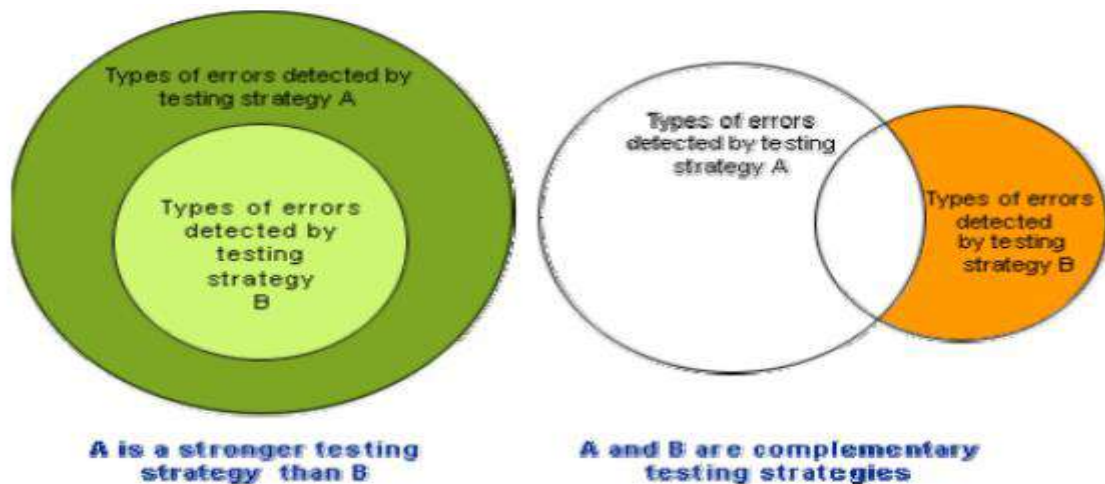
### **Summary of the Black-box test suite Design**

- Examine the input and output values of the program.
- Identify the equivalence classes.
- Pick the test cases corresponding to equivalence class testing and boundary value analysis

### **WHITE -BOX TESTING**

- White Box Testing is software testing technique in which internal structure, design and coding of software are tested to verify flow of input-output and to improve design, usability and security.
- In white box testing, code is visible to testers so it is also called Clear box testing, Open box testing, Transparent box testing, Code-based testing and Glass box testing.
- It is a test case design method that uses the control structure of the procedural design to derive test cases. It is the most widely utilized unit testing to determine all possible paths within a module, to execute all loops and to test all logical expressions. This form of testing concentrates on procedural detail.
- One white-box testing strategy is said to be stronger than another strategy, if all types of errors detected by the first testing strategy

is also detected by the second testing strategy, and the second testing strategy additionally detects some more types of errors. When two testing strategies detect errors that are different at least with respect to some types of errors, then they are called complementary. The concepts of stronger and complementary testing are schematically illustrated in fig.



## **DIFFERENT WHITE BOX METHODOLOGIES:**

**1.STATEMENT COVERAGE**

**2.BRANCH COVERAGE,**

**3.CONDITION COVERAGE,**

**4.PATH COVERAGE,**

**5.DATA FLOW BASED TESTING**

**6.AND MUTATION TESTING.**

### **STATEMENT COVERAGE**

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

statement is executed there is no way to determine whether an error exist in that statement unless a statement is executed, we cannot observe whether it causes failure due to some illegal memory access, wrong result computationetc.

Example:

Consider 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 returnx;  
    }
```

Design of test cases for the above program segment

Test case1	Statement executed
x=5,y=5	1,5,6
Test case2	Statement executed
x=5,y=4	1,2,3,5,6
Test case3	Statement executed
x=4,y=5	1,2,4,5,6

so the test set of the above algorithm will be  
{(x=5,y=5),(x=5,y=4),(x=4,y=5)}.

## BRANCH COVERAGE

In the branch coverage-based testing strategy, test cases are

designed to make each branch condition assume true and false value in turn. Branch testing is also known as edge testing, which is stronger than statement coverage testing approach.

Example : As the above algorithm contains two control statements such as while and if statement, so this algorithm has two number of branches. As each branch contains a condition, therefore each branch should be tested by assigning true value and false value respectively. So four number of test cases must be designed to test the branches.

Testcase1          x=6,y=6

Testcase2          x=6,y=7

Testcase3          x=8,y=7

Testcase4          x=7,y=8

so the test set of the above algorithm will be  
 $\{(x=6,y=6),(x=6,y=7),(x=8,y=7),(x=7,y=8)\}$ .

## CONDITION COVERAGE

In this structural testing, test cases are designed to make each component of a composite conditional expression assumes both true and false values. For example, in the conditional expression  $((C_1 \text{ AND } C_2) \text{ OR } C_3)$ , the components  $C_1, C_2$  and  $C_3$  are each made to assume both true and false values. Condition testing is a stronger testing strategy than branch testing and branch testing is a stronger testing strategy than the statement coverage- based testing.

## **PATH COVERAGE**

The path coverage-based testing strategy requires designing test cases such that all linearly independent paths in the program are executed at least once. A linearly independent path can be defined in the 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. The different numbered statements serve as nodes of the control flow graph (as shown in fig). An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node.

The CFG for any program can be easily drawn by knowing how to represent the sequence, selection, and iteration type of statements in the CFG. After all, a program is made up from these types of statements.

**Sequence:**

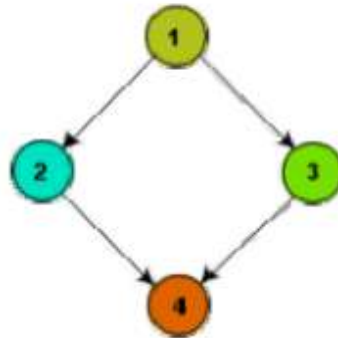
1.  $a=5;$
2.  $b=a^2-1;$



(a)

**Selection:**

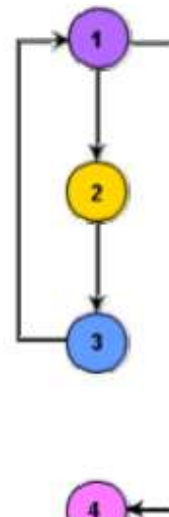
1.  $\text{if}(a>b)$
2.  $c=3;$
3.  $\text{else } c=5;$
4.  $c=c^*c;$



(b)

**Iteration:**

1.  $\text{while}(a>b)\{$
2.  $b=b-1;$
3.  $b=b^*a;\}$
4.  $c=a+b;$



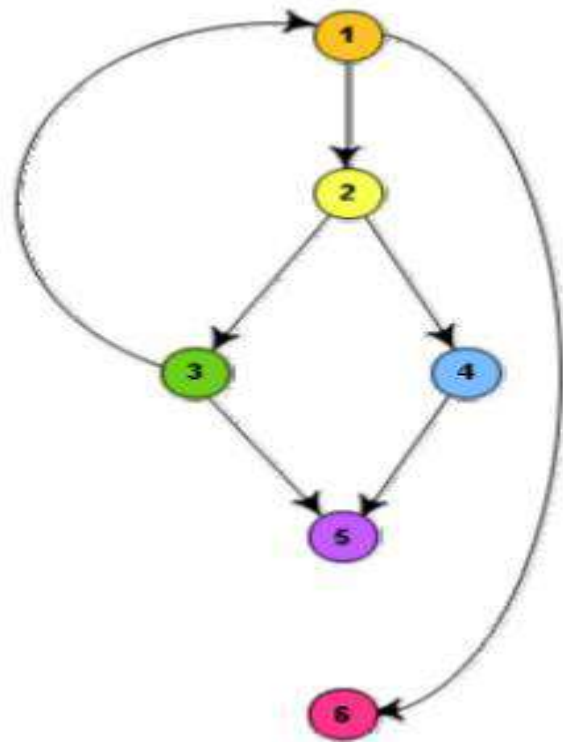
(c)

**Fig. 10.3:** CFG for (a) sequence, (b) selection, and (c) iteration type of constructs

```

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

```



(a) Example program

(b) Control Flow Graph

## 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. This is because, any path having a new node automatically implies that it has a new edge. Thus, a path that is sub path of another path is not considered to be a



linearly independent path.

## PATH

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

## MCCABE'S CYCLOMATIC COMPLEXITY METRIC

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

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

### 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:

$$V(G) = \text{Total number of bounded areas} + 1$$

For the CFG example shown in fig. 10.4, 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  $N+1$ .

## **DATA FLOW – BASED TESTING**

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

Consider a program  $P$ . For a statement numbered  $S$  of  $P$ , let  $DEF(S) = \{X \mid \text{Statement } S \text{ contains a definition of } X\}$ , and

$USES(S) = \{X \mid \text{Statement } S \text{ contains a use of } X\}$

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

The definition of variable  $X$  at statement  $S$  is said to be live at statement  $SI$ , If there exist a path from statement  $S$  to statement  $SI$  which doesn't contain any definition of  $X$ .

## **MUTATION TESTING**

In mutation testing, the software is first tested by using an initial

test suite built of from different white – box testing strategies. After the initial testing is complete, mutation testing is taken up. The idea behind mutation testing is to make a few arbitrary changes to a program at a time. Each time the program is changed, it is called a mutated program and the change effected is called a mutant. A mutated program is tested against the full test suite of the program. If there exists at least one test case in the test suite for which a mutant gives an incorrect result, then the mutant is said to be dead. If a mutant remains alive even after all the test cases have been exhausted, the test data is enhanced to kill themutant.

A major disadvantage of the mutation – based testing approach is that it is computationally very expensive since a large number of possible mutants can be generated.

Since mutation testing generates large mutants and requires us to each mutant with the full test suite. It is not suitable for manual testing.

## DEBUGGING

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

### Buffer Force Method

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

approach becomes more systematic with the use of a symbolic debugger because the values of different variables can be easily checked.

### Backtracking

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

### Cause Elimination Method

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

### Program Slicing

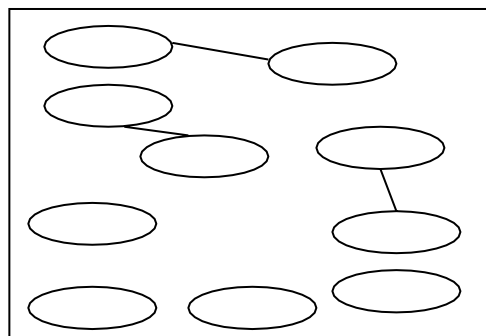
This technique is similar to back tracking. However, the search space is reduced by defining slices.

### Debugging Guidelines

- Debugging is often carried out by programmers based on their ingenuity.
- 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 many introduce new errors.

## Need for Integration Testing

The objective of integration testing is to test the module interfaces in order to ensure that there are no errors in the parameter passing, when one module invokes another module. During integration testing different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined



to realize the full system. After each integration step, the partially integrated system is tested.

Fig.6.5 Integration Testing

Anyone or a mixture of the following approaches can be used to develop the test plan:

- Big – bang approach
- Top – down approach
- Bottom – up approach
- Mixed approach

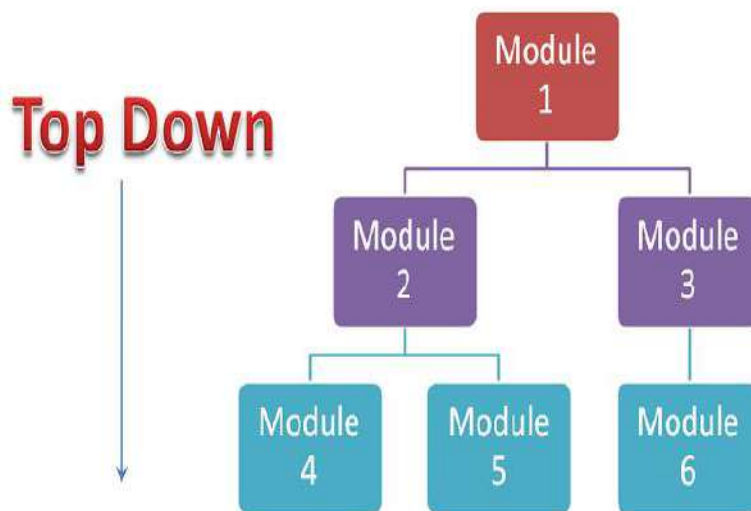
### Big – Bang Approach

In this approach, all the modules of the system are simply put together and tested. This technique is practicable only for small systems. The main problem with this approach is that once an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the

modules being integrated. Debugging errors reported during big-bang integration testing are very expensive.

### Top – Down Approach

Top-down integration testing starts with the main routine and one or two subordinate routines in the system. After the top-level 'skeleton' has been tested, the immediately subroutines of the 'skeleton' are combined with it and tested. Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test. A pure top-down integration does not require any driver routines. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, many times it may become difficult to exercise the toplevel routines in the desired manner since the lower-level routines perform several low-level functions such as I/O.



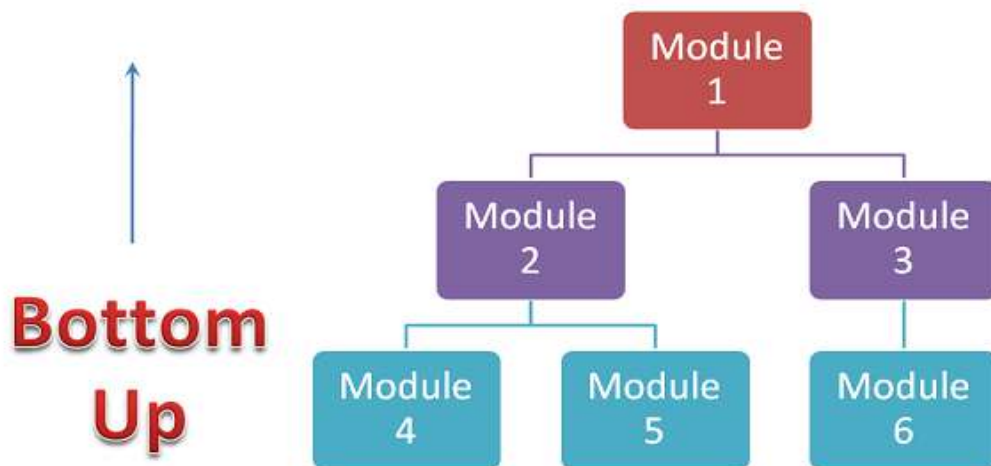
## Bottom – up Integration Testing

In bottom-up testing, each subsystem is tested separately and then the full system is tested. A subsystem might consist of many modules which communicate among each other through well-defined interfaces. The primary purpose of testing each subsystem is to test the interfaces among various modules making up the subsystem. Both control and data interfaces are tested. The test cases must be carefully chosen to exercise the interfaces in all possible manners. Large software systems normally require several levels of subsystem testing; lower-level subsystems are successively combined to form higher-level subsystems.

A principal advantage of bottom-up integration testing is that several disjoint subsystems can be tested simultaneously. In a pure bottom-up testing no stubs are required, only test-drivers are required.

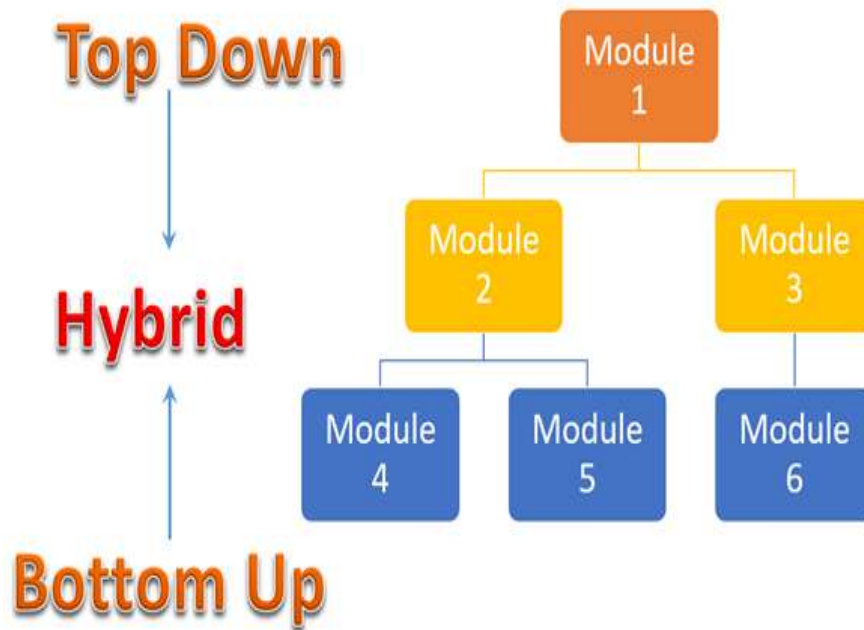
A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems. The extreme case corresponds to the big-bang approach





### Mixed Integration Testing

A mixed (also called sandwiched) integration testing follows a combination of topdown and bottom-up testing approaches. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready. The mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. In the mixed testing approaches, testing can start as and when modules become available. Therefore, this is one of the most commonly used integration testing approaches.



#### Phased vs. incremental testing

The different integration testing strategies are either phased or incremental. A comparison of these two strategies is as follows:

- In incremental integration testing, only one new module is added to the partial system each time.
- In phased integration, a group of related modules are added to the partial system each time.

#### System Testing:

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

- Alphatesting
- Betatesting
- Acceptancetesting

## Alpha Testing

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

## Beta testing

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

## Acceptance Testing

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

The system test cases can be classified into functionality and performancetest case. The functionality test are designed to check whether the software satisfies the functional requirements as documented in the SRS document. The performance tests test the conformance of to the system with the nonfunctional requirements of thesystem.

## Performance Testing

Performance testing is carried out to check whether the system meets thenon

– functional requirements identified in the SRS document. The types of performance testing to be carried out on a system depend on the different nonfunctional requirements of the system document in the SRS document. All performance tests can be considered as black – boxtests.

- Stress testing
- Volume testing
- Configuration testing

- Compatibility testing
- Regression testing
- Recovery testing
- Maintenance testing
- Documentation testing
- Usability testing

## Stress Testing

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

Stress testing is especially important for systems that usually operate below the maximum capacity but are severely stressed at some peak demand hours.

Example : If the nonfunctional requirement specification states that the response time should not be more than 20 seconds per transaction when 60 concurrent users are working, then during the stress testing the response time is checked with 60 users working simultaneously.

## Volume Testing

Volume testing, as the name suggests, is a testing done on high volumes of data. It belongs to a group of non-functional testing that is performed as part of performance-testing where a software product or application with high volume of data is tested, like huge number of input files, data records or heavy database table size in the system. It is also known as Flood 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.

- If you want to test the application with a database of a specific size, the database of the system should be expanded by adding more data to the system database until it meets the target.

Performing either of these activities or a related one will serve as an effective volume testing technique that will possibly give the best result.

### Configuration Testing

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

Configuration Testing is the process of testing the system under each configuration of the supported software and hardware. Here, the different configurations of hardware and software means the multiple operating system versions, various browsers, various supported drivers, distinct memory sizes, different hard drive types, various types of CPU etc.

#### **Various Configurations:**

##### **Operating System Configuration:**

Win XP, Win 7 32/64 bit, Win 8 32/64 bit, Win 10 etc.

##### **Database Configuration:**

Oracle, DB2, MySql, MSSQL Server, Sybase etc.

##### **Browser Configuration:**

IE 8, IE 9, FF 16.0, Chrome, Microsoft Edge etc.

Configuration testing is used to test system behavior in various

hardware and software configuration specified in the requirements.

### Compatibility Testing

Checking the functionality of an application on different software, hardware platforms, network, and browsers is known as compatibility testing.

Once the application is stable, we moved it to the production, it may be used or accessed by multiple users on the different platforms, and they may face some compatibility issues, to avoid these issues, we do one round of compatibility testing.

### Types of Compatibility testing

Following are the types of compatibility testing:

- **Software**
- **Hardware**
- **Network**
- **Mobile**

### Software

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

And, we have two types of version compatibility testing, which are as follows:

**Forward Compatibility Testing:** Test the software or application on the new or latest versions.

**For example:** Latest Version of the platforms (software)



**Win 7 → Win 8 → Win 8.1 → Win 10**

**Backward Compatibility Testing:** Test the software or application on the old or previous versions.

**For example:**

Window XP → Vista → Win 7 → Win 8 → Win 8.1

And different browsers like **Google Chrome**, **Firefox**, and **Internet Explorer**, etc.

### **Hardware**

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

### **Mobile**

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

### **Network**

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

## **Regression Testing**

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

### **When to do regression testing?**

- When a new functionality is added to the system and the

code has been modified to absorb and integrate that functionality with the existing code.

- When some defect has been identified in the software and the code is debugged to fix it.
- When the code is modified to optimize its working.

### Recovery Testing

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

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

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

#### What Is Maintenance Testing?

Once software has been launched, it runs for years. But during the course of its natural life, this software needs to be upgraded or enhanced and sometimes even migrated to other hardware, in order to add more years to its successful run. The testing that is conducted during the enhancement stage or migration cycle of already deployed software is known as Maintenance Testing.

### Documentation Testing

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

### Usability Testing

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

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

Sometimes the customer might specify the maximum number of allowable errors that may be present in the delivered system. These are often expressed in terms of maximum number of allowable errors per line of source code. Error seed can be used to estimate the number of residual errors in a system. Error seeding, as the name implies, seeds the code with some known errors. In other words, some artificial errors are introduced into the program artificially. The number of these seeded errors detected in the course of the standard testing procedure is determined. These values in conjunction with the number of unseeded errors detected can be used to predict:

- The number of errors remaining in the product.

- The effectiveness of the testing strategy.

Let  $N$  be the total number of defects in the system and let  $n$  of these defects be found by testing.

Let  $S$  be the total number of seeded defects, and let  $s$  of these defects be found during testing.  $n/N = s/S$  or  $N = S \times n/s$

Defects still remaining after testing =  $N - n = n \times (S - s)/s$

Error seeding works satisfactorily only if the kind of seeded errors matches closely with the kind of defects that actually exist. However, it is difficult to predict the types of errors that exist in a software. Error seeding is useful only to a moderate extent.

## **UNIT-VII**

### **UNDERSTANDING THE IMPORTANCE OF S/W RELIABILITY**

**DEFINITIONS OF SOFTWARE RELIABILITY** Software reliability is defined as the probability of failure-free operation of a software system for a specified time in a specified environment. The key elements of the definition include probability of failure-free operation, length of time of failure-free operation and the given execution environment. Failure intensity is a measure of the reliability of a software system operating in a given environment. Example: An air traffic control system fails once in two years.

- Reliability of a software product essentially denotes its trustworthiness or dependability. Alternatively, reliability of a software product can also be defined as the probability of the product working “correctly” over a given period of time.
- It is obvious that a software product having a large number of defects is unreliable.
- It is also clear that the reliability of a system improves, if the number of defects in it is reduced. However, there is no simple relationship between the observed system reliability and the number of latent defects in the system. For example, removing errors from parts of a software which are rarely executed makes little difference to the perceived reliability of the product.
- It has been experimentally observed by analyzing the behavior of a large number of programs that 90% of the execution time of a typical program is spent in executing only 10% of the instructions in the program. These most used 10% instructions are often called the core of the program. The rest 90% of the program statements are called non-core and are executed only for 10% of the total execution time. It

therefore may not be very surprising to note that removing 60% product defects from the least used parts of a system would typically lead to only 3% improvement to the product reliability. It is clear that the quantity by which the overall reliability of a program improves due to the correction of a single error depends on how frequently is the corresponding instruction executed.

- Thus, reliability of a product depends not only on the number of latent errors but also on the exact location of the errors.
- Apart from this, reliability also depends upon how the product is used, i.e. on its execution profile. If it is selected input data to the system such that only the “correctly” implemented functions are executed, none of the errors will be exposed and the perceived reliability of the product will be high.
- On the other hand, if the input data is selected such that only those functions which contain errors are invoked, the perceived reliability of the system will be very low.

### **Factors Influencing Software Reliability**

- A user’s perception of the reliability of a software depends upon two categories of information.
  - The number of faults present in the software.
  - The way users operate the system. This is known as the operational profile.

### **Reasons for software reliability being difficult to measure**

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

- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.

- The perceived reliability of a software product is highly observer dependent
- The reliability of a product keeps changing as errors are detected and fixed.

### **HARDWARE RELIABILITY VS. SOFTWARE RELIABILITY**

Reliability behavior for hardware and software are very different.

For example, hardware failures are inherently different from software failures. Most hardware failures are due to component wear and tear.

A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix hardware faults, one has to either replace or repair the failed part.

On the other hand, a software product would continue to fail until the error is tracked down and either the design or the code is changed. For this reason, when a hardware is repaired its reliability is maintained at the level that existed before the failure occurred;

whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug introduces new errors).

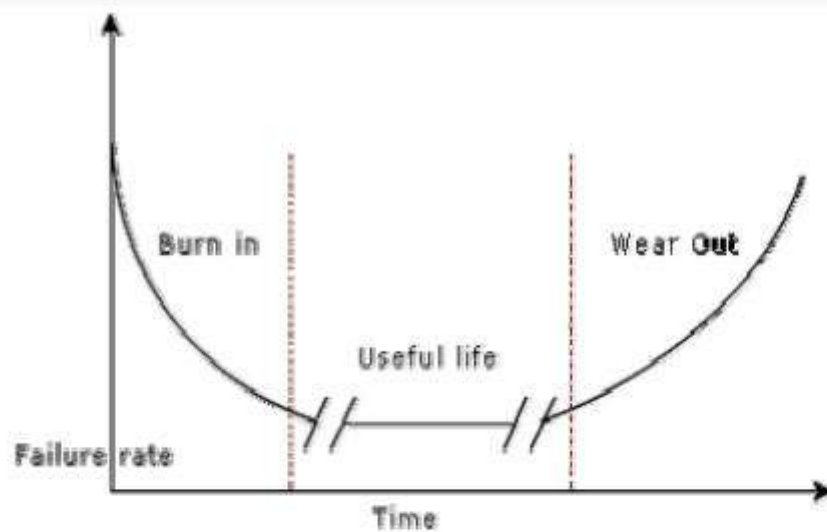
To put this fact in a different perspective, hardware reliability study is concerned with stability (for example, inter-failure times remain constant).

On the other hand, software reliability study aims at reliability growth (i.e. inter-failure times increase). The change of failure rate over the product lifetime for a typical hardware and a software product are sketched in fig. .

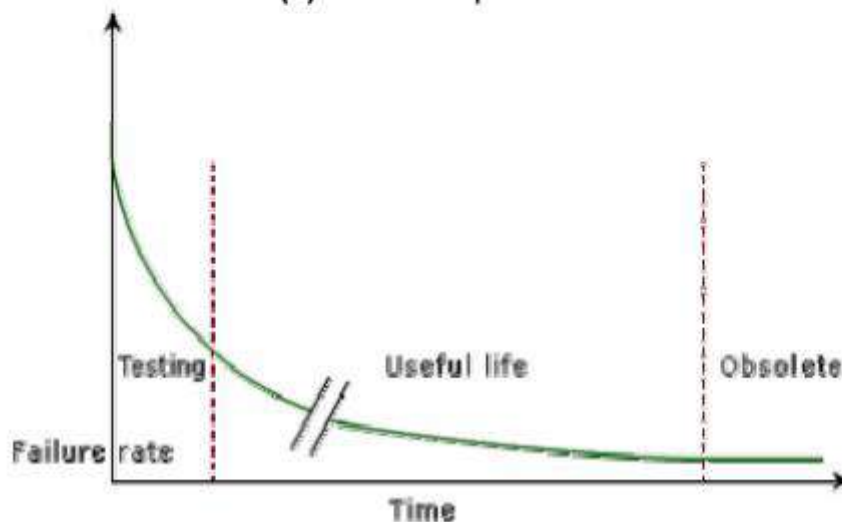
For hardware products, it can be observed that failure rate is high initially but decreases as the faulty components are identified and removed. The system then enters its useful life. After some time (called product life time) the components wear out, and the failure rate increases. This gives the plot of hardware reliability over time its characteristics “bath tub” shape. On the other hand, for software the failure rate is at it’s highest during integration and test. As the system is tested, more and more errors are identified and removed resulting in reduced failure rate. This error removal continues at a slower pace during the useful life of the product. As the software becomes



obsolete no error corrections occurs and the failure rate remains unchanged.



(a) Hardware product

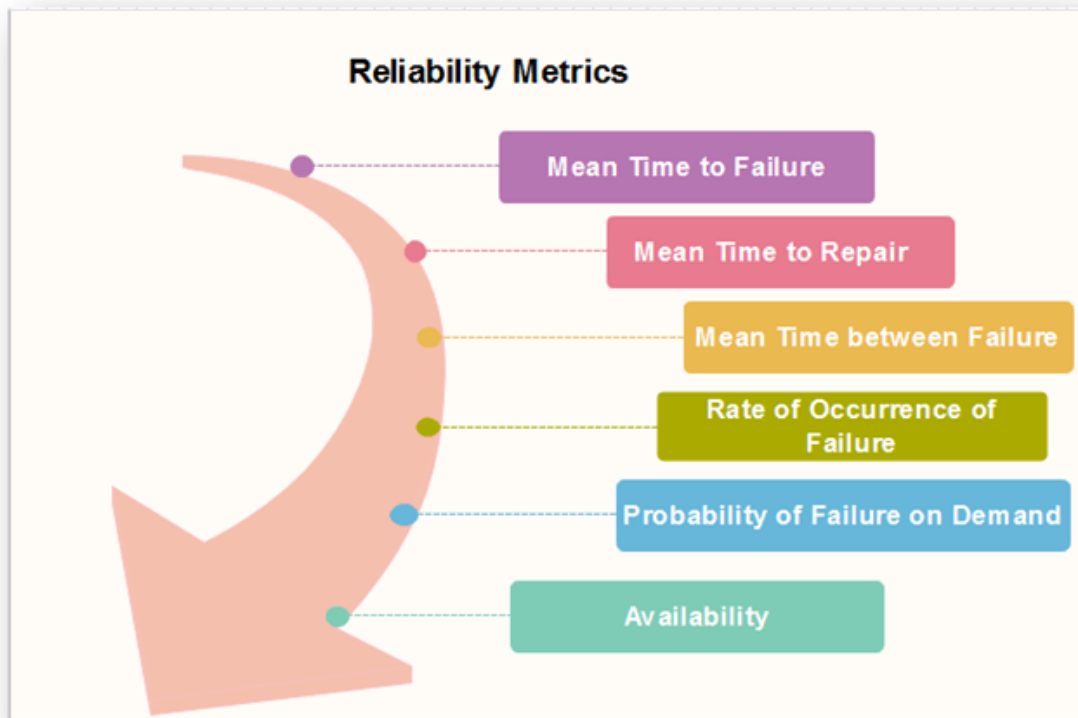


(b) Software product

## RELIABILITY METRICS

Reliability metrics are used to quantitatively expressed the reliability of the software product.

Some reliability metrics which can be used to quantify the reliability of the software product are as follows:



The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the SRS (software requirements specification) document. In order to be able to do this, some metrics are needed to quantitatively express the reliability of a software product. A good reliability measure should be observer-dependent, so that different people can agree on the degree of reliability a system has. For example, there are precise techniques for measuring performance, which would result in obtaining the same performance value irrespective of who is carrying out the performance measurement. However, in practice, it is very difficult to formulate a precise reliability measurement technique. The next base case is to have measures that correlate with reliability. There are six reliability metrics which can be used to quantify the reliability of software products.

- **RATE OF OCCURRENCE OF FAILURE (ROCOF)**

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

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

MTTF is the average time between two successive failures, observed over a large number of failures. To measure MTTF, we can record the failure data for  $n$  failures. Let the failures occur at the time instants  $t_1, t_2, \dots, t_n$ .

Then, MTTF can be calculated as

$$\sum_{i=0}^n \frac{t_{i-1} - t_i}{(n - 1)}$$

It is important to note that only run time is considered in the time measurements, i.e. the time for which the system is down to fix the error, the boot time, etc are not taken into account in the time measurements and the clock is stopped at these times.

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

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

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

MTTF and MTTR can be combined to get the MTBR metric:  $MTBF = MTTF + MTTR$ . Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours. In this case, time measurements are real time and not the execution time as in MTTF.

- **PROBABILITY OF FAILURE ON DEMAND (POFOD).**

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

time measurements. POFOD measures the likelihood of the system failing when a service request is made. For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure. V

- **AVAILABILITY**

Availability of a system is a measure of how likely shall the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs. This metric is important for systems such as telecommunication systems, and operating systems, which are supposed to be never down and where repair and restart time are significant and loss of service during that time is important.

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

- Transient. Transient failures occur only for certain input values while invoking a function of the system.
- Permanent. Permanent failures occur for all input values while invoking a function of the system.
- Recoverable. When recoverable failures occur, the system recovers with or without operator intervention.
- Unrecoverable. In unrecoverable failures, the system may need to be restarted.
- Cosmetic. These classes of failures cause only minor irritations, and do not lead to incorrect results.

An example of a cosmetic failure is the case where the mouse button has to be clicked twice instead of once to invoke a given function through the graphical user interface

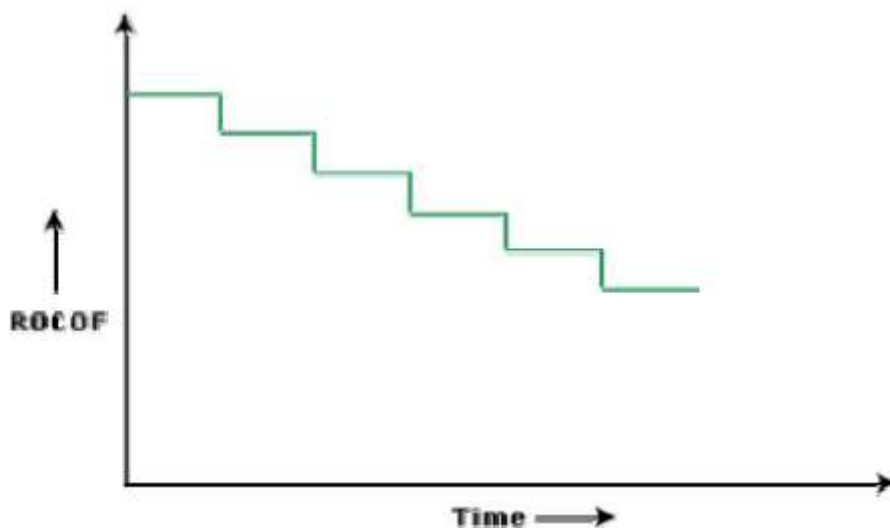
## **RELIABILITY GROWTH MODELS**

A reliability growth model is a mathematical model of how software reliability

improves as errors are detected and repaired. A reliability growth model can be used to predict when (or if at all) a particular level of reliability is likely to be attained. Thus, reliability growth modeling can be used to determine when to stop testing to attain a given reliability level. Although several different reliability growth models have been proposed, in this text we will discuss only two very simple reliability growth models.

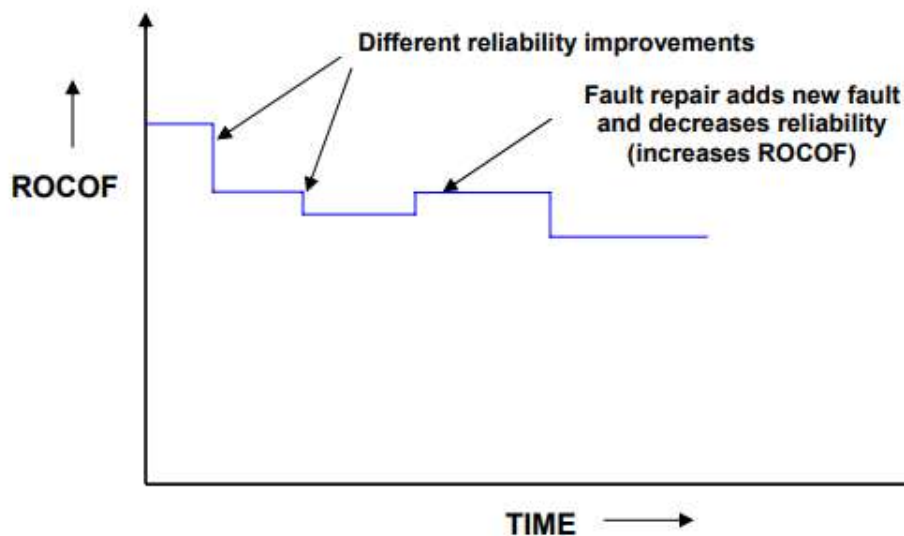
### **JELINSKI AND MORANDA MODEL**

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.. 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.



### **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. 13.3). It treats an error's contribution to reliability improvement to be an independent random variable having Gamma distribution. This distribution models the fact that error corrections with large contributions to reliability growth are removed first. This represents diminishing return as test continues.



## **SOFTWARE QUALITY**

Traditionally, a quality product is defined in terms of its fitness of purpose. That is, a quality product does exactly what the users want it to do. For software products, fitness of purpose is usually interpreted in terms of satisfaction of the requirements laid down in the SRS document. Although “fitness of purpose” is a satisfactory definition of quality for many products such as a car, a table fan, a grinding machine, etc. – for software products, “fitness of purpose” is not a wholly satisfactory definition of quality. To give an example, consider a software product that is functionally correct. That is, it performs all functions as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally correct, we cannot consider it to be a quality product. Another example may be that of a product which does everything that the users want but has an almost incomprehensible and unmaintainable code. Therefore, the traditional concept of quality as “fitness of purpose” for software products is not wholly satisfactory. The modern view of a quality associates with a software product several quality factors such as the following:

- **Portability:** A software product is said to be portable, if it can be easily made to work in different operating system environments, in different machines, with other software products, etc.
- **Usability:** A software product has good usability, if different categories of users (i.e., both expert and novice users) can easily invoke the functions of the product.
- **Reusability:** A software product has good reusability, if different modules of the product can easily be reused to develop new products.
- **Correctness:** A software product is correct, if different requirements as specified in the SRS document have been correctly implemented.
- **Maintainability:** A software product is maintainable, if errors can be easily corrected as and when they show up, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

### **SOFTWARE QUALITY MANAGEMENT**

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

#### **Managerial Structure and Individual Responsibilities.**

A quality system is actually the responsibility of the organization as a whole. However, every organization has a separate quality department to perform several quality system activities. 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 systems have rapidly evolved over the last five decades. Prior to World War II, the usual method to produce quality products was to inspect the finished products to eliminate defective products. Since that time, quality systems of organizations have undergone through four stages of evolution as shown in the fig. The initial product inspection method gave way to quality control (QC). Quality control focuses not only on detecting the defective products and eliminating them but also on determining the causes behind the defects. Thus, quality control aims at correcting the causes of errors and not just rejecting the products. The next breakthrough in quality systems was the development of quality assurance principles. The basic premise of modern quality assurance is that if an organization's processes are good and are followed rigorously, then the products are bound to be of good quality. The modern quality paradigm includes guidance for recognizing, defining, analyzing, and improving the production process

Total quality management (TQM) advocates that the process followed by an organization must be continuously improved through process measurements. TQM goes a step further than quality assurance and aims at continuous process improvement. TQM goes beyond documenting processes to

optimizing them through redesign. A term related to TQM is Business Process Reengineering (BPR). BPR aims at reengineering the way business is carried out in an organization. From the above discussion it can be stated that over the years the quality paradigm has shifted from product assurance to process assurance (as shown in fig. ).

