P.K.A.I.E.T

We Excel Where We Stand

SOFTWARE ENGINEERING

Fifth Semester

COMPUTER SCIENCE & ENGG.

Prepared By: Mr Prasanta ku.Satapathy

(Sr.Lecturer)

CHAPTER-01
Introduction to Software Engineering

Articles to be covered

Program vs. Software product

PROGRAM

● Programs are developed by individuals for their personal use.
● They are generally small in size and have limited functionality.
● The author of a program himself uses and maintains his program, these usually do not have a good user interface and lack proper documentation.
● Program consists of a set of instructions which is a combination of source code and object code.

Software product

● Software products have multiple users and therefore should have a good user interface, proper operating procedures, and good documentation support.

● Since a **Software product** has a large no of users, it must be properly designed, carefully implemented and properly tested.
● Generally software products are too large and they cannot be developed by a single programmer.
● Therefore, **Software products** are developed by a group of software engineers.

Emergence of Software Engineering

Software engineering techniques have evolved over many years which resulted in a series of innovations and accumulation of experience about writing good quality programs. Innovations and programming experiences which have contributed to the development of software engineering are as follows.

Early Computer Programming

Early commercial computers were very slow as compared to today's standard computers. Even simple processing tasks took more computation time on those computers. No wonder that programs at that time very small in size and lacked sophistication. Those programs were usually written in assembly languages. Program lengths were typically limited to about a few

hundreds of lines of monolithic assembly code. Every programmer writes the programs in his own style.

High-Level Language Programming

Computers become faster with the introduction of semiconductor technology. With the availability of more powerful computers, it became possible to solve larger and more complex problem. High Level languages such as FORTRAN, ALGOL and COBOL were introduced. This considerably reduced the effort required to develop software products and helped programmers to write larger programs. However, the software development style was limited to sizes of around a few thousands of lines of source code.

Control Flow-Based Design

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 program's 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 of 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.

Computer Systems Engineering

- Computer systems engineering is one of the most popular engineering fields, with the number of professionals growing steadily.
- A computer systems engineer develops, tests, and evaluates software and personal computers by combining their knowledge of engineering, computer science, and math analysis.
- Contrary to popular belief, computer systems engineers do not merely engineer computer technology. Rather, they are expected to fully comprehend how that technology is used on a wider scale to suit both personal and professional requirements.

Software Life Cycle Models

The goal of software engineering is to provide models and processes that lead to the production of well-documented maintainable software. A life cycle model prescribes the different activities that need to be carried out to develop a software product and the sequencing of these activities. A software life cycle is the series of identifiable stages that a software product undergoes during its lifetime. It also captures the order in which these activities are to be undertaken. A software life cycle model is a descriptive and diagrammatic representation of the software life cycle. The various

phases of software life cycle or Software Development Life Cycle (SDLC) are:

- Preliminary Investigation
- Software Analysis
- Software Design
- Software Testing
- Software Maintenance

A software life cycle model is referred to as software process model.

Classical Waterfall model

- This model is called a linear sequential model.
- This model suggests a systematic approach to software development.
- The project development is divided into sequences of well-defined phases.
- It can be applied for long-term projects and well understood product requirements.
- 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

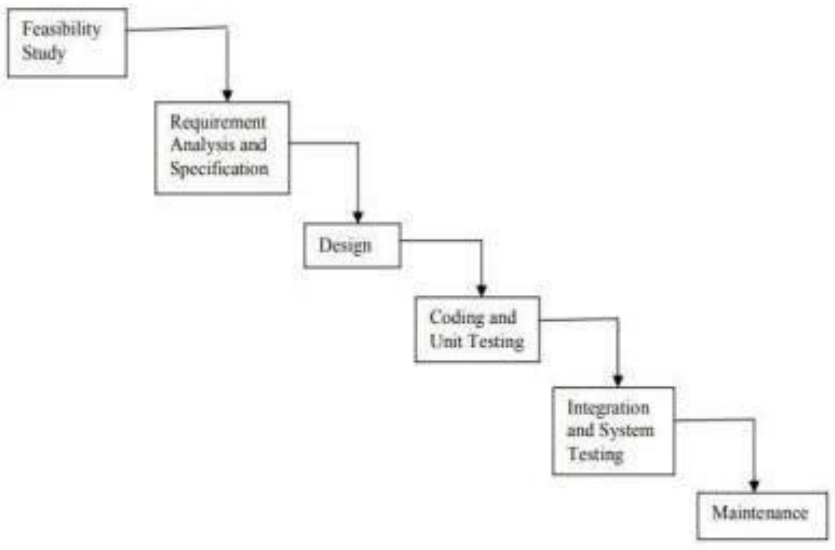

Feasibility Study
Requirement Analysis and Specification
Design
Coding and Unit Testing
Integration and System Testing
Maintenance

Fig. 1.1 Classical Waterfall Model

The phases starting from the feasibility study to the integration and system testing phases are known

as the development phases. All these activities
are performed in a set of sequences without skip or repeat. None of the activities can be revised once closed and the results are passed to the next step for use.

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 Feasibility Study Design Coding and Unit Testing Maintenance Requirement Analysis and Specification Integration and System Testing Fig. 1.1 Classical Waterfall Model 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? These phases capture the important requirements of the customer, also formulate all the different ways in which the problem can be solved are identified.

- 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. Each function can be characterized by the input data, the processing required on the input data and the output data to be produced. The non-functional requirements identify the performance requirements, the required standards to be followed etc. The 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: first a structured analysis of the requirements specification is carried out, second structured design activity. 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. Structured design consists of two main activities: architectural design (also called high level design) and detailed design (also called low level design). High level design involves decomposing the system into modules, representing the interfaces and the invocation relationships among the modules. Detailed design deals with data structures and algorithm of the modules.

- 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. Unit testing also involves a precise definition of the test cases, testing criteria and management of test cases.

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. During each integration step previously planned modules are added to the partially integration system and the resultant system is tested.
- Finally, after all the modules have been successfully integrated and tested system testing is carried out. The goal of system testing is to ensure that the developed system confirms to its requirements laid out in the SRS document.

- System testing usually consists of three different kinds of testing activities:
  - α –testing: α testing is the system testing performed by the development team.
  - β –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:

Iterative Waterfall model

- The classical waterfall model is an idealistic one since it assumes that no development error is ever committed by the engineers during any of the life cycle phases.
- Suppose a defect is detected at testing phase the engineers need to go back to the phase where the defect had occurred and correct the work done during that phase and the subsequent phases to correct the defect and its effect on the later phases.
- In any practical software development work it is not possible to strictly follow the classical waterfall model.
- Feedback paths are needed in the classical waterfall model from every phase to its preceding phases. It may not always be possible to detect all error in the same phase in which they occur.
- The feedback paths allow for correction of the errors committed during a phase, as and when these are detected. If during testing a design error is identified then the feedback path

allows the design to be reworked and the changes to be reflected in the design documents. However observe that there is no feedback path to the feasibility stage. This means that the feasibility study error can not be corrected.

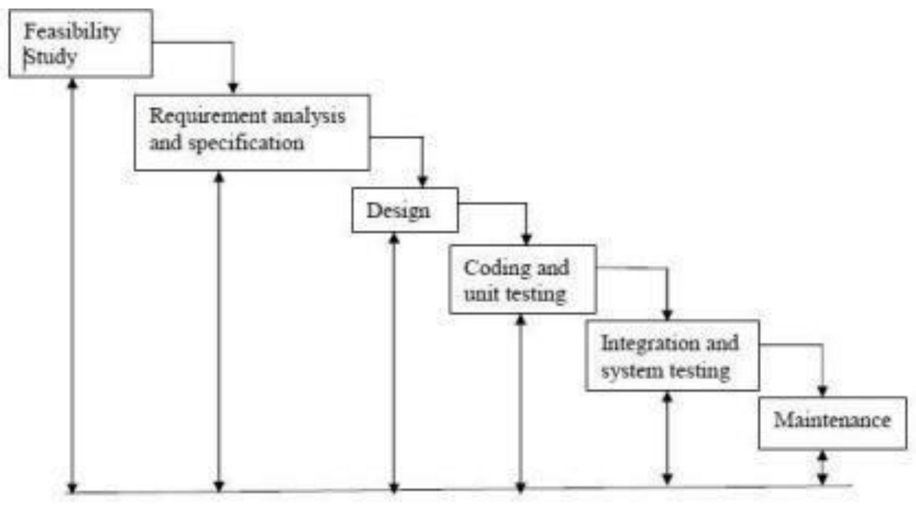

Feasibility Study
Requirement analysis and specification
Design
Coding and unit testing
Integration and system testing
Maintenance

Fig. 1.2 Iterative waterfall Model

Though errors are inevitable in almost every phase of development, it is desirable to detect these errors in the same phase in which they occur. This can reduce the effort required for correcting bugs. The principle of detecting errors as close to there points of introduction as possible is known as **phase containment of errors.** This is an important software engineering principle.

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 being applied.
- A prototype usually exhibits limited functional capabilities, low reliability and inefficient

performance compared to the actual software. A developed prototype can help engineers to critically examine the technical issues associated with product develop

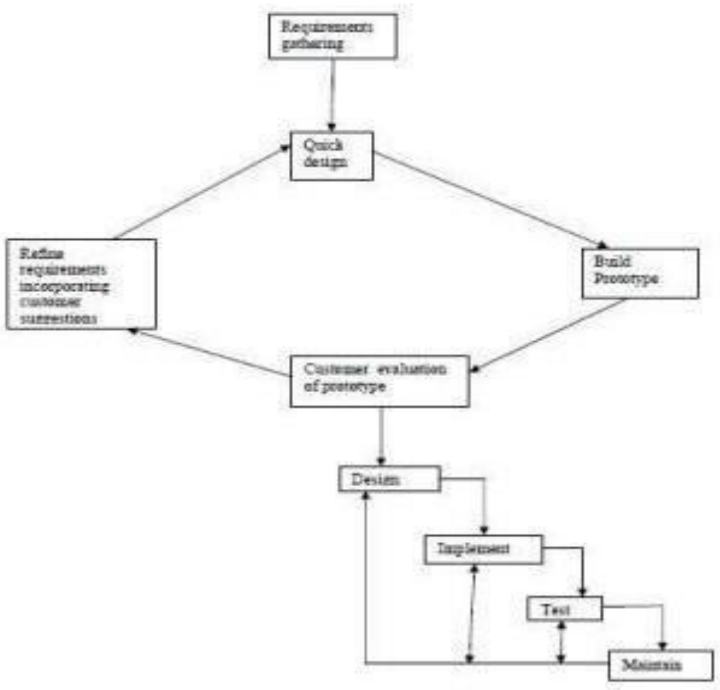

Requirements gathering
Quick design
Refine requirements incorporating customer suggestions
Build Prototype
Customer evaluation of prototype
Design
Implement
Test
Maintain

Fig. 1.3 Prototyping Model of Software Development

- 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 Prototyping Model of Software Development Requirements gathering Quick Refine requirements incorporating customer suggestions Build Prototype Customer evaluation of prototype design,Implement,Test, Maintain feedback to the developers regarding the prototype: what is correct, what needs to be modified, what is missing, what is not needed
- 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 the software requirement and specification (SRS) document, the prototype is discarded and the actual system is then developed using the iterative waterfall approach.

Disadvantage

- Prototyping is often not used, because development costs may become large.
- This model requires extensive participation and involvement of the customer, which is not always possible.

Evolutionary model

This life cycle model is also referred as the successive versions model and the incremental model. In this life cycle model the software is first broken down into several modules or functional units which can be incrementally constructed and delivered.

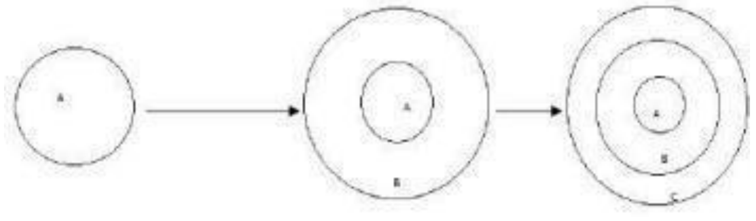


Fig. 1.4 Evolutionary model of software development

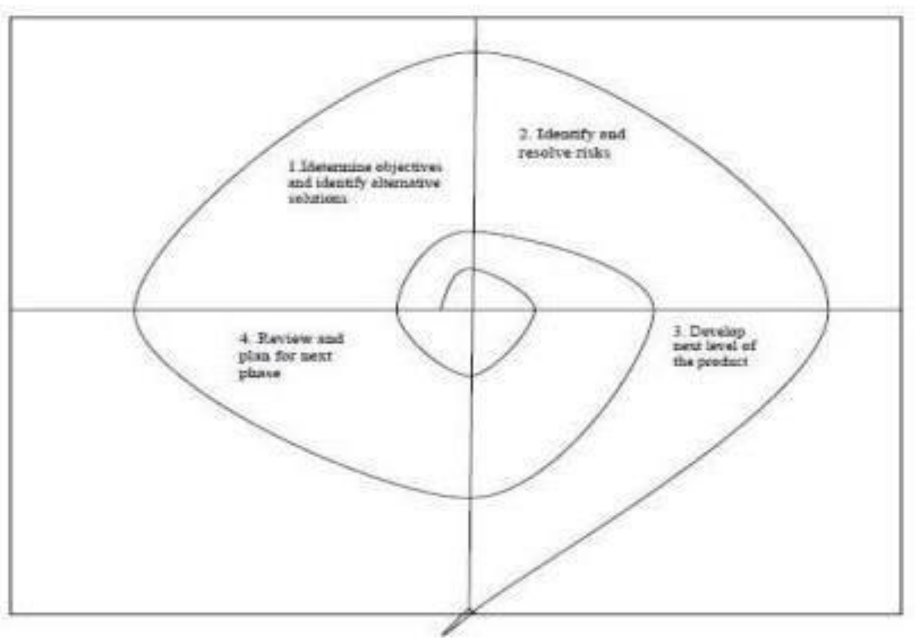
- A, B, C are modules of a software product that are incrementally developed and delivered.
- The development team first develops the core modules of the system. That is basic requirements are addressed but many supplementary features remain undelivered.
- The initial product is refined into increasing levels of capability by adding new functionalities in successive versions.
- Each evolutionary version may be developed using an interactive waterfall model of development

Disadvantage

The main disadvantage of the successive versions model is that for most practical problems it is difficult to divide the problem into several functional units which can be incrementally implemented and delivered. The evolutionary model is normally useful for only very large products.

Spiral Model

- The spiral model also known as the spiral life cycle model is a systems development life cycle model used in information technology.
- This model of development combines the features of the prototyping model, the waterfall model and other models.
- The diagrammatic representation of this model appears like a spiral with many loops.


1.Determine objectives and identify alternative solutions
2. Identify and resolve risks
3. Develop next level of the product
4. Review and plan for next phase

(Spiral Model of Software Development)

● Exact number of phases through which the product is developed in this model is not fixed.

● The number of phases varies from one project to another.

Each phase in this model is split into four sectors or quadrants:

1. **Planning**: Identifies the objectives of the phase and the alternative solutions possible for the phase and constraints.

2. **Risk analysis**: Analyze alternatives and attempts to identify and resolve the risks involved.

· 3. **Development**: Product development and testing product. ·

4. **Assessment**: Customer evaluation.

● During the first phase planning is performed, risks are analysied, prototypes are built and customers evaluate the prototype.

● During the second phase a second prototype is evolved by a fourfold procedure: evaluating the first prototype in terms of its strengths, weaknesses and risks, defining the requirements of the second prototype, constructing and testing the second prototype.

● The existing prototype is evaluated in the same manner as was the previous prototype and if necessary another prototype is developed.

● After several iterations along the spiral, all risks are resolved and the software is ready for development.

● At this point, a waterfall model of software development is adopted.

- The radius of the spiral at any point represents the cost incurred in the project till then and the angular dimension represents the progress, made in the current phase.
- In the spiral model of development, the project team must decide how exactly to structure the project into phases.
- The most distinguishing feature of this model is its ability to handle risks. The spiral model uses prototyping as a risk reduction mechanism and also retains the systematic step-wise approach of the waterfall model.

Spiral Model Strengths

- Provides early indication of risks, without much cost.
- Critical high-risk functions are developed first.
- Early and frequent feedback from users.
- Cumulative costs assessed frequently.

Spiral Model Weaknesses

- The model is complex.
- Risk assessment expertise is required.
- May be hard to define objectives.
- Spiral may continue indefinitely.
- Time spent planning, resetting objectives, doing risk analysis and
- prototyping may be excessive.

Chapter - 2
Software Project Management

Articles to be covered

Responsibility of Project Manager
Project Planning
    Metrics for Project size estimation(LOC and FP)
Project Estimation Techniques
COCOMO Models, Basic, Intermediate and complete
Scheduling
Organization and Team structure
Staffing
Risk Management
    Configuration 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.

There are many software engineers involved in the development of a software product. The primary job of the project manager is to ensure that the project is completed within budget and on schedule.

Responsibilities of 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 takes 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.

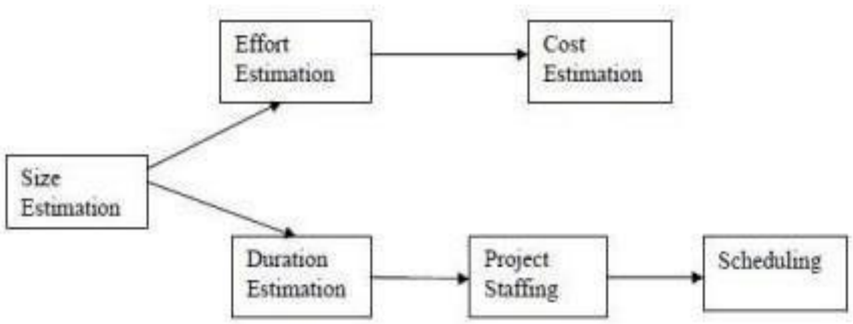


Fig. 2.2 Precedence Ordering among Planning Activities

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.

Project Size Estimation Metrics, Line Of Control (LOC) and Function Point Metric (FP)

➢ The size of a project is obviously not the number of bytes that thesource code occupies.

➢ The project size is a measure of the problem complexity in terms of the effort andtime required to develop the product.

➢ Two metrics are widely used to estimate size:

- · Lines of Code (LOC)
- · Function Point (FP)

Lines Of Code (LOC)

➢ LOC can be defined as the number of delivered lines of code in software excluding the comments and blank lines.

➢ LOC depends on the programming language chosen for the project.

➢ The exact number of the lines of code can only be determined after the project is complete since less information about the project is available at the early stage of development.

➢ In order to estimate the LOC count at the beginning of a project, project managers usually divide the problem into modules and each modules into sub modules and a so on until the sizes of the different leaf level modules can be approximately predicted.

Disadvantages:

➢ LOC is language dependent. A line of assembler is not the same as a line of COBOL.

➢ LOC metrics penalizes use of higher level programming languages, code reuseetc.

➢ 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 Metric

➢ Function Points measure software size by quantifying the functionality providedto user based solely on logical design and functional specifications

➢ Function point analysis is a method of quantifying the size and complexity of a softwaresystem in terms of the functions that the system delivers to the user

➢ It is independent of the computer language, development methodology,technology or capability of the project team used to develop the application.

➢ Function point analysis is designed to measure business applications(not scientific applications) .

➢ Function points are independent of the language, tools, or methodologies usedfor implementation

➢ Function points can be estimated early in analysis and design Since function points are based on the system user's external view of the system, non-technical users of the software system have a better understanding of what function points are measuring.

Objectives of Function Point Counting

➢ Measure functionality that the user requests and receives

➢ Measure software development and maintenance independently of technology used for implementation

Function point metric estimates the size of a software product directly from the problem specification.

The different parameters are:
.

1. Number Of Inputs:
Each data item input by the user is counted.
2. Number Of Outputs:
The outputs refers to reports printed, screen outputs, error messages produced etc.
3. Number Of Inquiries:
It is the number of distinct interactive queries which can be made by the users.

4.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.

5.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.

Heuristic Techniques

- ❖ Heuristic techniques assume that the relationships among the differentproject 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 = c1* ed1

Where e is a characteristics of the software, c1 and d1 are constants.
A multivariable cost estimation model takes the following form:

Estimated Resource = c1 * e1
d1 + c2 * e2
d2 + ...

Where e1, e2 ... are the basic characteristics of the software.
c1, c2, d1, d2... are constants.

Analytical Estimation Techniques

- ❖ Analytical estimation techniques derive the required results starting withcertain basic assumptions regarding the project.

- ❖ This technique does have a scientific basis.

❏ Halstead’s Software Science an Analytical EstimationTechniques

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
N1 be the total number of operators used in the program
N2 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.

Length and Vocabulary

The length of a program as defined by Halstead, quantifies the total usage of all operations and operands in the program.
Thus, length $N = N1 + N2$
The program vocabulary is the number of unique operators and operands used in the program.
Thus, program vocabulary $\eta = \eta 1 + \eta 2$

Program Volume
The length of a program depends on the choice of the operators and operands used. $V = N \log 2 \; \eta$
The program volume V is the minimum number of bits needed to encode the Program.

In fact, to represent η different identifiers uniquely, we need at least $\log 2 \; \eta$ bits. We

need N log2 η 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.

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

Empirical Estimation Techniques

- ❖ Cost estimation is a part of the planning stage of any engineering activity.Forany new software project, it is necessary to know how much it will cost to develop and how much development time it will take.
- ❖ Cost in a project is due to the requirements for software, hardware andhuman resources.
- ❖ Hardware resources such as computer time, terminal time and memory required for the project, software resources include the tools and compilersneeded during development.
- ❖ Cost estimates can be made either top-down or bottom-up.
- ❖ Top-down estimation first focuses on system level costs such as the computing resources and personal required to develop the system, qualityassurance system integration, training.
- ❖ Bottom-up cost estimation first estimates the cost to develop each moduleor subsystem.
- ❖ Those costs are combined to arrive at an overall estimate.

Two popular empirical estimation techniques

1. 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 an expert makes an educated guess of the problem sizeafter analyzing the problem thoroughly.
- ❖ The expert estimates the cost of the different modules or subsystems andthen combines them to arrive at the overall estimate.
- ❖ An expert making an estimate may not have experience and knowledge ofall 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 expertsand 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 anonymouslyand submit it to the coordinator.
- ❖ They may ask questions to the coordinator, but they do not discusstheir estimates with one another.
- ❖ The coordinator prepares and distributes a summary of the

estimator’s responses and includes any unusual rationales noted bythe estimators.

❖ Based on this summary, the estimators re-estimate. This process is iteratedfor several rounds.

❖ No group discussion is allowed during the entire process.

COCOMO: A Heuristic Estimation Technique

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, and 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.

Semi Detached: 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:

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

Basic COCOMO

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

Effort = a1 × (KLOC)a2 PM
Tdev = b1 × (Effort) b2 Months

Where
(i) KLOC is the estimated size of the software product expressed in Kilo Lines of Code, (ii)

a1, a2, b1, b2 are constants for each category of software products,

(iii) Tdev 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).

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 thesize 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:

1. Product attributes
2. Computer attributes

3. Personnel attributes
4. 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 smallersubsystem.
- ❖ 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 outphase wise.
- ❖ Detailed COCOMO provides estimated phase-wise efforts and duration ofphase 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, integrationand system testing. Total effort is estimated separately. This approach reduces the margin of error in the final estimate.

Scheduling

❖ Scheduling the project tasks is an important project planning activity.

❖ Scheduling involves deciding which tasks would be taken up when.

❖ In order to schedule the project activities, a software project manager needs todo the following.

i) Identify all the tasks necessary to complete the project.
ii) Break down larger tasks into a logical set of small activates which would be assigned to different engineers.
iii) Create the work break down structure and to find the dependency among the activates. Dependency among the different activates determines the order in which the different activates would be carried out.
iv) Establish the most likely estimates for the time durations necessary to complete the activities.
v) Resources are allocated to each activity. Resource allocation is typically done using a Gantt chart.
vi) Plan the starting and ending dates for various activities. The end of each activity is called a milestone.
Vii) Determine the critical path.

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

- The first step in scheduling a software project involves identifying all the tasks necessary to complete the project. Next, the large tasks are broken down into logical set of small activities which would be assigned to different engineers.
- After the project manager has broken down the task and created the work breakdown structure, he has to find the dependency among the activities. Dependency among the different activities determines the order in which the different activities would be carried out. If an activity A requires the results of another activity B, then activity A must be scheduled after activity B. The task dependencies define a partial ordering among tasks.
- Once the activity network representation has been worked out, resources are allocated to each activity. Resource allocation is typically done using a Gantt chart.
- After resource allocation is done, a Project Evaluation and ReviewTechnique chart

representation is developed. The PERT chart representation is suitable for program monitoring and control.

Use of Work Breakdown Structure, Activity Networks,Gantt Chart and PERT in Scheduling

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 labelled 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 undertaken.

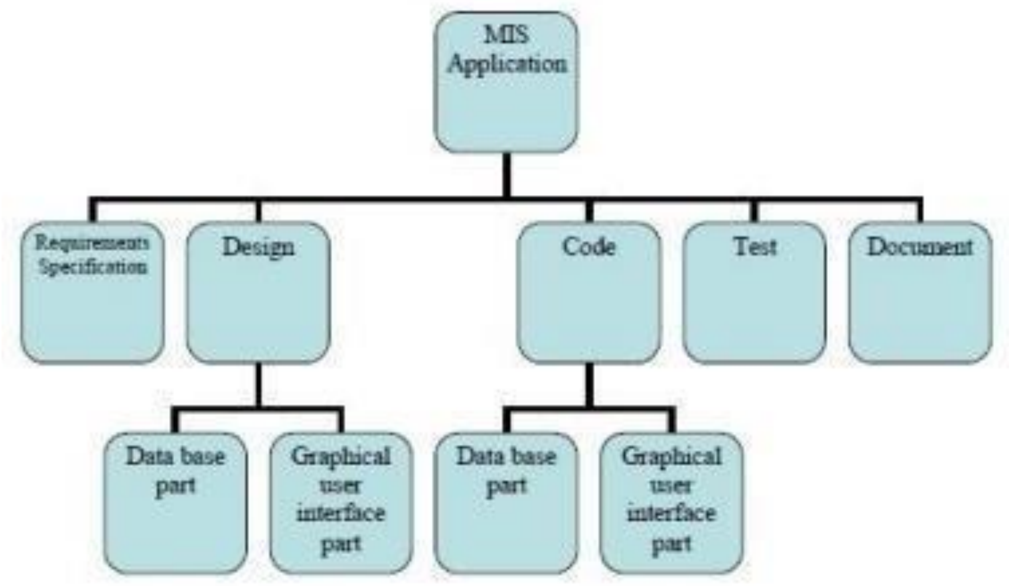

MIS Application
Requirements Specification
Design
Code
Test
Document
Data base part
Graphical user interface part
Data base part
Graphical user interface part

Fig. 2.3 Work breakdown structure of an MIS problem

- The task is broken down into a large number of small activities; these activities can be distributed to a large number of engineers. Thus it becomes possible to develop the product faster.
- Therefore, to be able to complete a project in the least amount of time the manager needs to break large tasks into smaller subtasks, expecting to find more parallelism. In scheduling the manager decide the order in which to do these tasks.
- Two general scheduling techniques are Gantt Charts and PERT Charts.

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.

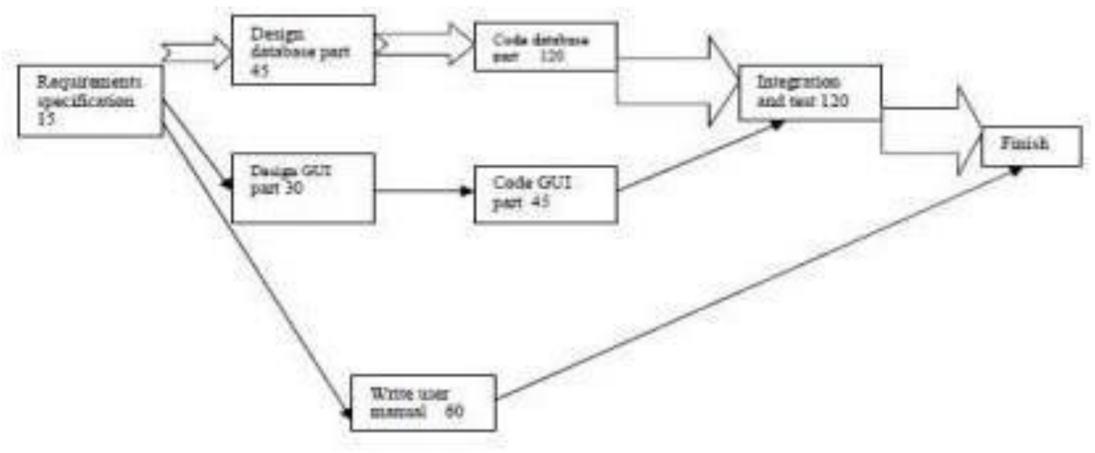


Fig. 2.4 Activity Network representation of the MIS problem

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

- 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 the 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

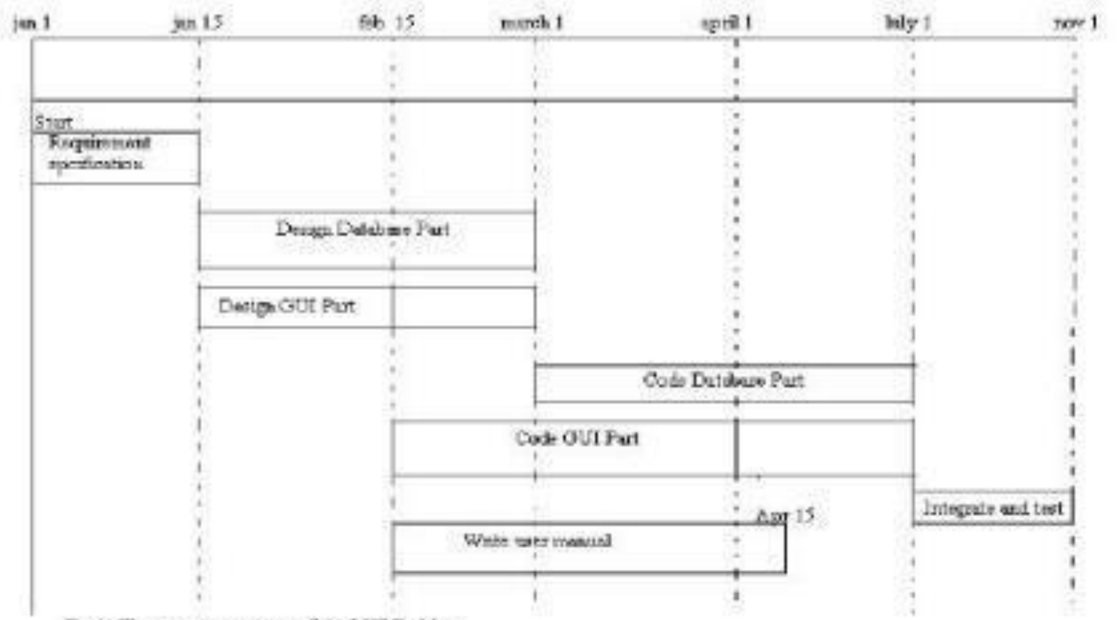
path.

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

Gantt Chart

● Gantt charts are a project control technique that can be used for several purposes including scheduling, budgeting and resource planning. Gantt Charts are mainly used to allocate resources to activities. A Gantt chart is a special type of bar chart where each bar represents an activity.

● The bars are drawn against a time line. The length of each bar is proportional to the duration of the time planned for the corresponding activity.



Gantt Chart representation of the MIS Problem

Fig. 2.5 Gantt Chart Representation of the MIS Problem

● In the Gantt Chart the bar consists of a write part and a shaded part. The shaded part of the bar shows the length of time each task is estimated to take. ●

The white part shows the slack time, that is the latest time by which a task must be finished.

PERT (Project Evaluation and Review Technique) Charts

- PERT controls time and cost during the project and also facilities finding the right balance between completing a project on time and cost during the project and also facilitates finding the right balance between completing a project on time and completing it within a budget.

- A PERT Chart is a network of boxes (or circles) and arrows. The boxes represent activities and the arrows are used to show the dependencies of activities on one another.

- The activity at the head of an arrow cannot start until the activity at the tail of the arrow is finished. The boxes in a PERT Chart can be decorated with starting and ending dates for activities.

- PERT Chart is more useful for monitoring the timing progress of activities.

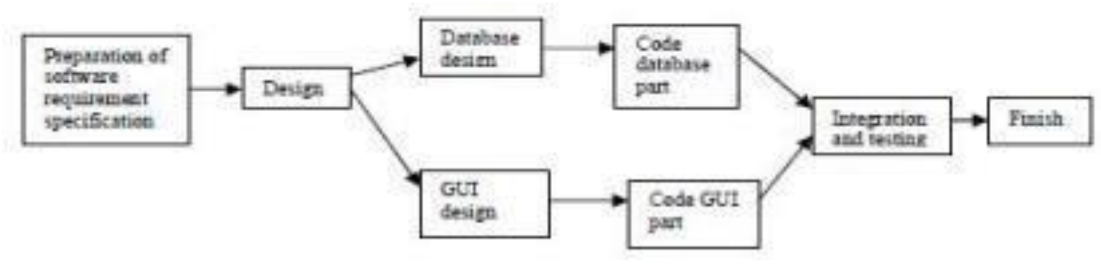


Fig.2.6 PERT Chart representation of the MIS problem

PERT Chart shows the interrelationship among the tasks in the project and identifies critical path of the project.

Organization and Team structure

- There are essentially two broad ways in which a software development organization can be structured:
  - function format
  - project format.
- In the project format, the development staff are divided based on the project for which they

work. In the functional format, the development staff are divided based on the functional group to which they belong to .

- The different projects below engineers from functional groups for specific phases of the projects and return them to their functional group upon completion of the phase.

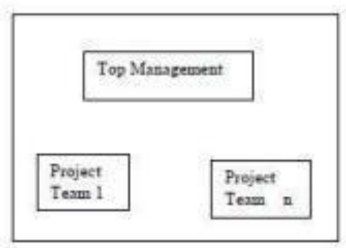


Fig. Project Organization

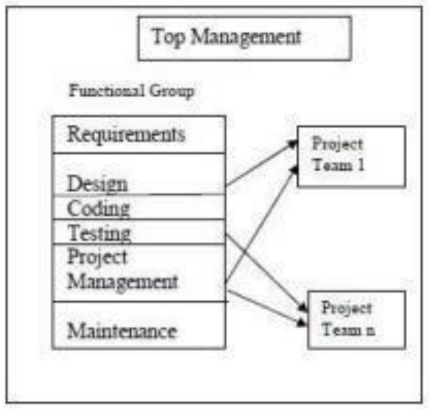


Fig.2.7 Functional organization

- In the functional format, different teams of programmers perform different phases of a project.

- For example, one team might do the requirements specification, another do the design, and so on.

- The partially completed product passes from one team to another as the product evolves. Therefore, the functional format requires considerable communication among the different teams because the work of one team must be clearly understood be team must be clearly understood by the subsequent teams working on the project
- In the project format, a set of engineers are assigned to the project at the start of the project and they remain with the project till the completion of the project. Thus, the same team carries out all the life cycle activities.

- Obviously, the functional format requires more communication among teams than the project format, because one team must understand the work done by the previous teams. The main advantages of a functional organization are:

1. Ease of staffing
2. Production of good quality documents
3. Job specialization
4. Efficient handling of the problems associated with manpower turnover

- The functional organisation allows engineers to become specialists in their particular roles, e.g. requirements analysis, design, coding, testing, maintenance etc. the functional organisation also provides an efficient solution to the staffing problem.

- A project organisation structure forces the manager to take in almost a constant number of engineers for the entire duration of the project.

Team Structure

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

Three format team structures are:

- 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.

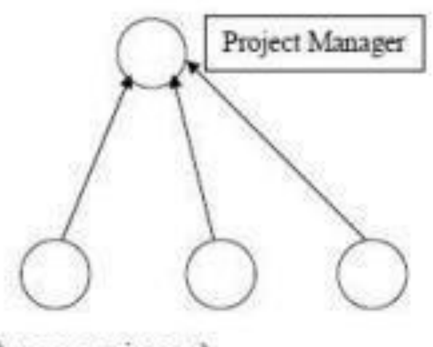


Fig. 2.8 Chief programmer team structure

- The chief programmer provides an authority. The chief programmer team leads to lower team morale, since the team members 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 intellectualgrasp 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.

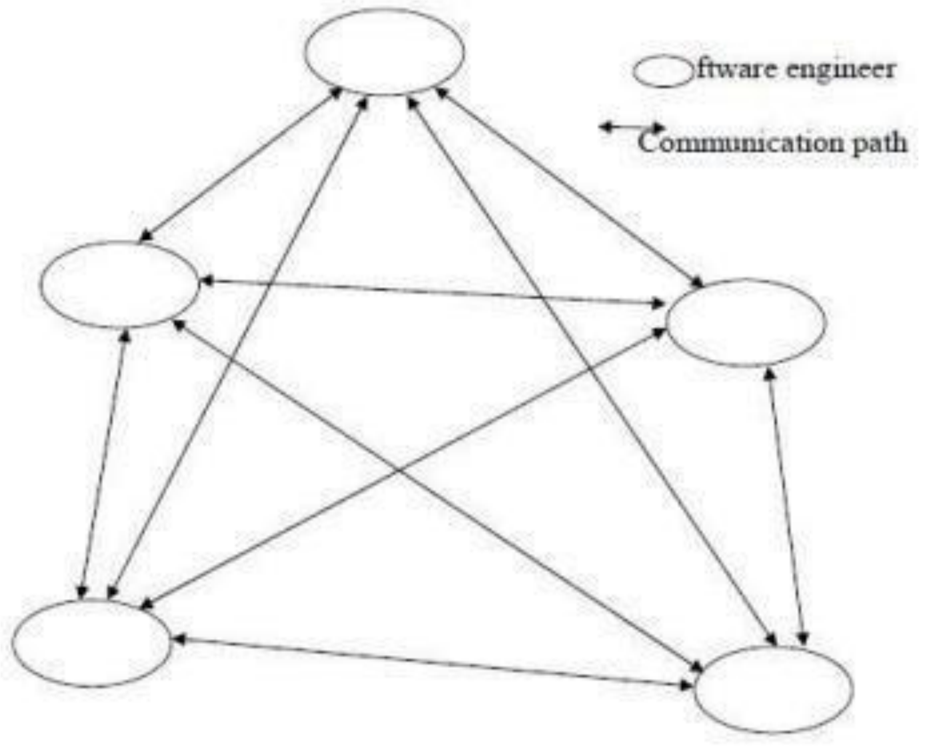


Fig.2.9 Democratic team structure

● 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 ideas from both the democratic organization and the chief programmer organization. This team organization incorporates both hierarchical reporting and democratic set-up.

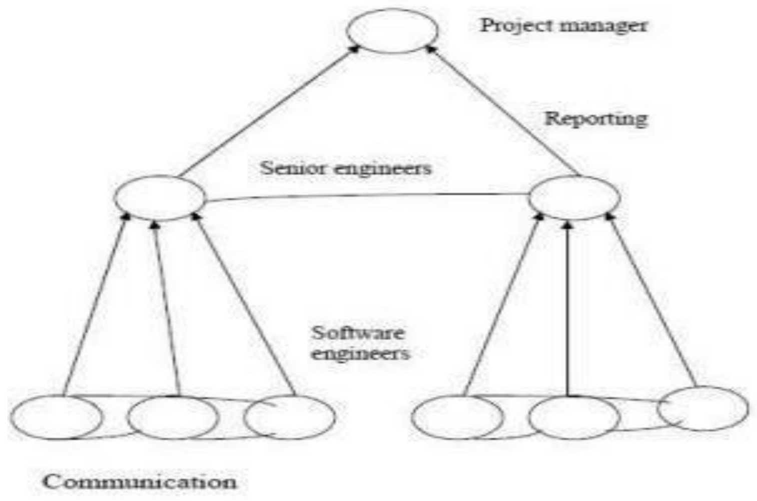


Fig.2.10 Mixed team structure

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

- Each democratic set-up at the programmer level attempts to find solution to a single part. This team attempts to find solution to a single part.

- This team structure is extremely popular and is being used in many software development companies.

Staffing

- Jensen Model for Staffing Level Estimation
- Jensen model is very similar to Putnam model. However, it attempts to soften the effect of schedule compression on effort to make it applicable to smaller and medium sized projects. Jensen proposed the equation:

L=Cte td K1/2

Where Cte is the effective technology constant, td is the time to develop the software, and K is the effort needed to develop the software.

Risk Management

- Risk management is an emerging area that aims to address the problem of identifying and managing the risk associated with a software project.

- Risk in a project is the possibility that the defined goals are not met. The basic motivation of having risk management is to avoid heavy looses.

- Risk is defined as an exposure to the chance of injury or loss. That is risk implies that there is possibility that something negative may happen.

- In the content of software project, negative implies that there is an adverse effect on cost, quantity or schedule. Risk management aims at reducing the impact of all kinds of risk that might affect a project.

Risk management consist of three essential activities:

- · Risk identification
- · Risk assessment
- · Risk containment

Risk Identification

A project can get affected by a large variety of risks. Risk identification identifies all the different risks for a particular project. In order to identify the important risks which might affect a project, it is necessary to categorize risk 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 risk concern potential design, implementation, interfacing, testing, and maintenance problem. Technical risks also include incomplete specification, changing specification, technical uncertainly.

Most technical risks occur due the 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 risks items. For risks assessment, each risk should be rated in two ways:

CHAPTER-3.0
Requirement Analysis and specification

Articles to be covered

Requirements gathering and analysis

Software Requirements Specification

Contents of SRS

Characteristics of Good SRS

Organization of SRS

Techniques for representing complexing logic

Requirements gathering and analysis

● Requirement analysis is a Software engineering task that bridges the gap between system level requirements engineering and software design. ● Requirement analysis provides software designers with a representation of system information, function, and behavior that can be translated to data, architectural, and component-level designs.

● Software requirement analysis may be divided into five areas of effort: ➢ Problem recognition

➢ Evaluation and synthesis

➢ Modeling

➢ Specification

➢ Review

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 clearlyunderstand the exact requirements of the customer.

➢ The analyst should understand the problems:

·

❑ 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 is the data output required of the system?

❑ · What are the complexities that might arise while solving the problem?

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

Software Requirements 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 very different purposes. Some of the important categories of users of the SRS document and their needs are as follows.

➢ Users, customers and marketing personnel: The goal of this set of audience is to ensure that the system as described in the SRS document will meet their needs. ➢ The software developers refer to the SRS document to make sure that they develop exactly what is required by the customer.

➢ Test Engineers: Their goal is to ensure that the requirements are understandable from a functionality point of view, so that they can test the software and validate 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 the functionalities of the system. A clear knowledge of the functionalities can help them to understand the design and code.

Contents of SRS

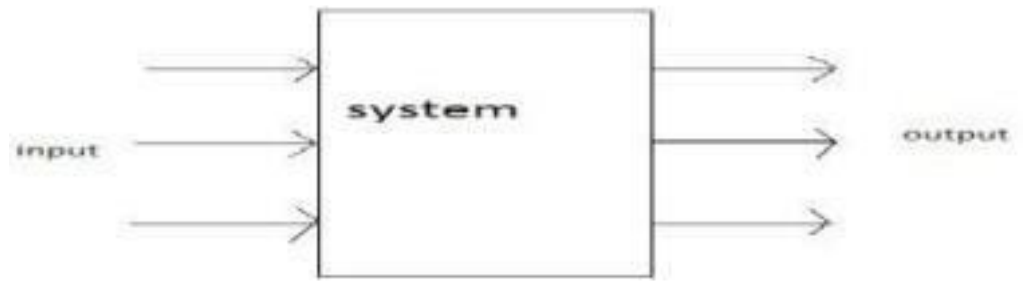
An SRS document should clearly document:

➢ · Functional Requirements

➢ · Nonfunctional Requirements

➢ · Goals of implementation

- 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 non-functional requirements are also known as quality requirements. The non-functional requirements deal with the characteristics of the system that cannot be expressed as functions.

- Examples of non-functional requirements include aspects concerning maintainability, portability and usability, accuracy of results.
- Non-functional requirements arise due to user requirements, budget constraints, organizational policies and soon.
- The goals of the implementation part of the SRS document gives some general suggestions regarding development. This section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future.

Characteristics of Good SRS

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

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

Block-box View: It should specify what the system should do. The SRS document should specify the external behavior of the system and not discuss the implementation issues. The SRS should specify the externally visible behavior of the system. [For this reason the SRS document is called the block-box specification of a system.]

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

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

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

Organization of SRS

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

1. Goals of implementation
   - ★ Functional requirements
   - ★ Nonfunctional Requirements
   - ★Behavioural Description

(a) System States
(b)Events and Actions

Possible Short Questions with answers

Chapter 4
Software Design

Articles to be covered

What is a Good Software Design

The goodness of a design is dependent on the targeted application. Different characteristics of a software design are:

• Correctness: A good design should correctly implement all the functionalities of the system.
• Understandability: A good design should be easily understandable.
• Efficiency: A good design solution should adequately address resource, time and cost optimization issues.
• Maintainability: A good design should be easy to change.

Cohesion and coupling

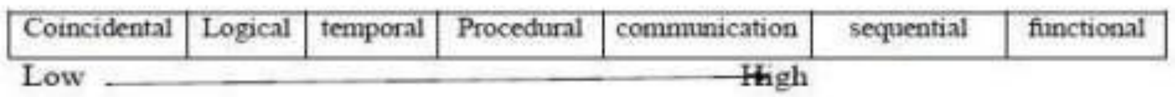
The primary characteristics of a neat module decomposition are high cohesion and low coupling

1. Cohesion

• Cohesion is a measure of the strength of the relationship between responsibilities of the components of a module.

• A module is said to be highly cohesive if its components are strongly related to each other by some means of communication or resource sharing or the nature of responsibilities.

Classification of Cohesiveness
There are seven types or levels of cohesion.



Coincidental	Logical	temporal	Procedural	communication	sequential	functional

Fig. 4.1 Classification of Cohesion

Coincidental is the worst type of cohesion and functional is the best cohesion.

1. **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.

• The different functions of the module carry out. The different unrelated activities are issuing of librarian leave request.

2. **Logical Cohesion**

• A module is said to be logically cohesive, if all elements of the module perform similar operations.
• For example, consider a module that consists of a set of print functions to generate various types of output reports such as salary slips annual reports etc.

3. **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.
• For example, consider the situation: when a computer is booted, several functions need to be performed.
• These include initialization of memory and devices, loading the operating system etc.
• When a single module performs all these tasks, then the module can be said to exhibit temporal cohesion.

4. **Procedural Cohesion**

• A module is said to possess procedural cohesion, if the set of functions of the module are executed one after the other, though these functions may work entirely different purposes and operate on different data.
• For example, in an automated teller machine(ATM),member-card validation is followed by

• personal validation by personal identification number and following this, the request option menu is displayed.

5. Communication Cohesion

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

6. Sequential Cohesion

A module is said to possess sequential cohesion, if the different functions of the module execute in a sequence, and the output from one function is input to the next in the sequence.

7. Functional Cohesion

- A module is said to possess functional cohesion, if different function of the module cooperate to complete a single task.
- The functions issue-book (), return-book (), query-book () and find borrower () together manage all activities concerned with book lending.

Coupling

- The coupling between two modules indicates the degree of interdependence between modules.
- Two modules with high coupling are strongly interconnected and thus dependent on each other.
- Two modules with low coupling are not dependent on one another. ”Uncoupled” modules have no interconnections, they are completely independent.

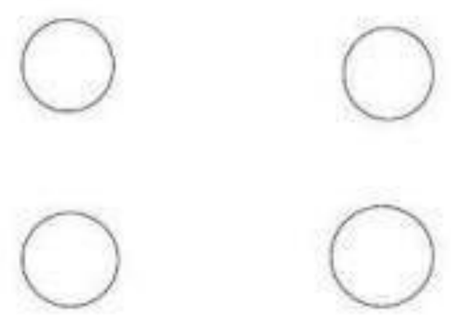


Fig. 4.2 Uncoupled: No Dependencies

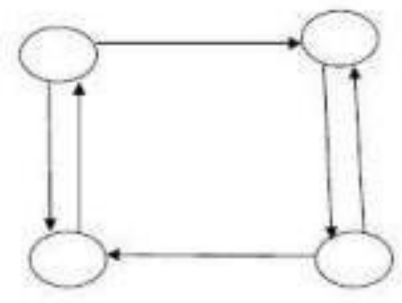


Fig. 4.3 Loosely coupled: some dependencies

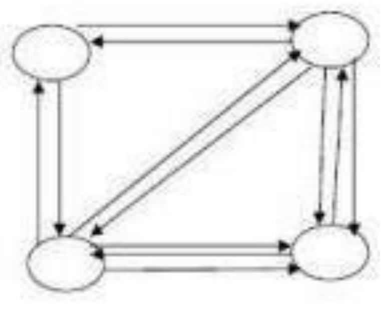


Fig. 4.4 Highly coupled: many dependencies

A good design will have low coupling. Coupling is measured by the number of interconnections between modules. Coupling increases as the number of calls between modules increases.

Different types of coupling are:

Data	Stamp	Control	Common	Content

Fig. 4.5 Classification of Coupling

1 . Data Coupling

- It is a type of loose coupling and combines modules by passing some parameters from one module to another.
- The parameters that are passed are usually atomic data type of programming language.
  Eg an integer, a float, a character etc.
- This data item should be problem related and not used for control purposes.

2. Stamp Coupling

Two modules are stamp coupled, if they communicate using a composite data item such as a structure in C.

3. Control Coupling

Module A and B are said to be control coupled if they communicate by passing of control information.

4. **Common Coupling**
Two modules are common coupled, if they share some global data items.

5. **Content coupling**
Content coupling exist between two modules, if their code is shared.eg. a branch from one module into another module.

Neat Arrangement

It should neatly arrange the modules in a hierarchy. e.g. tree-like diagram

(i)Layered solution
(ii) Low fan out
(iii) Abstraction

Software Design Approaches

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

1. 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 consists of the following subfunctions:
· assign-membership-number
· create-member-record
· print-bill
Each of these sub functions may be split into more detailed sub functions and so on.

2. 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 Flo Diagram (DFD).

Data Flow Diagram

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

Symbols used in DFD

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

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

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

Designing DFD

- A DFD model of a system graphically represent how each input data is transformed to its corresponding output data through a hierarchy of DFDs.
- A DFD start with the most abstract definition of the system (lowest level) and at each higher level DFD, more details are successively introduced.
- The most abstract representation of the problem is also called the context diagram.
- Context Diagram
- The context diagram represents the entire system as a single bubble. The bubble is labelled according to the main function of the system.
- The various external entities with which the system interacts and the data flows occurring between the system and the external entities are also represented.
- The data input to the system and the data output from the system are represented as incoming and outgoing arrows.

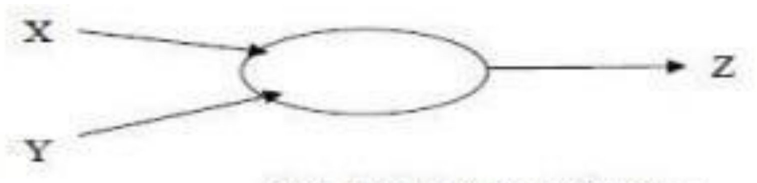


Fig. 4.6 Context Diagram

Level 1 DFD

- The level 1 DFD usually contains between 3 and 7 bubbles.
- 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 a bubble in the Level 1 DFD.

Decomposition

- Each bubble in the DFD represents a function performed by the system.
- The bubbles are decomposed into sub functions at the successive level of the DFD.
- 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 last stage

Developing DFD model of a System

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

➢ Registration module
➢ Examination module
➢ Result generation module

Registration module:

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

Examination module:

a) Assignments : Each subject has an associated assignment, which is compulsory and should be submitted by the student before a specified date.
b) Theory Papers : The theory papers can be core or elective. Core papers are compulsory papers, while in elective papers students have a choice to select.
c) Practical papers: The practical papers are compulsory and every semester has practical papers.

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

Data Flow Diagram

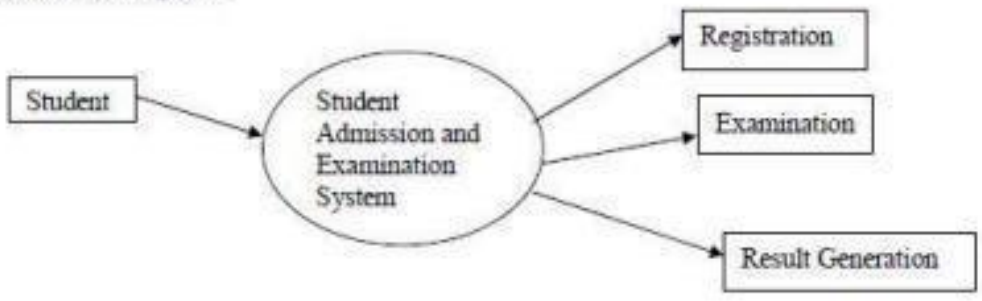


Fig. 4.7 Level 0 DFD or Context Diagram

Level 1 DFD

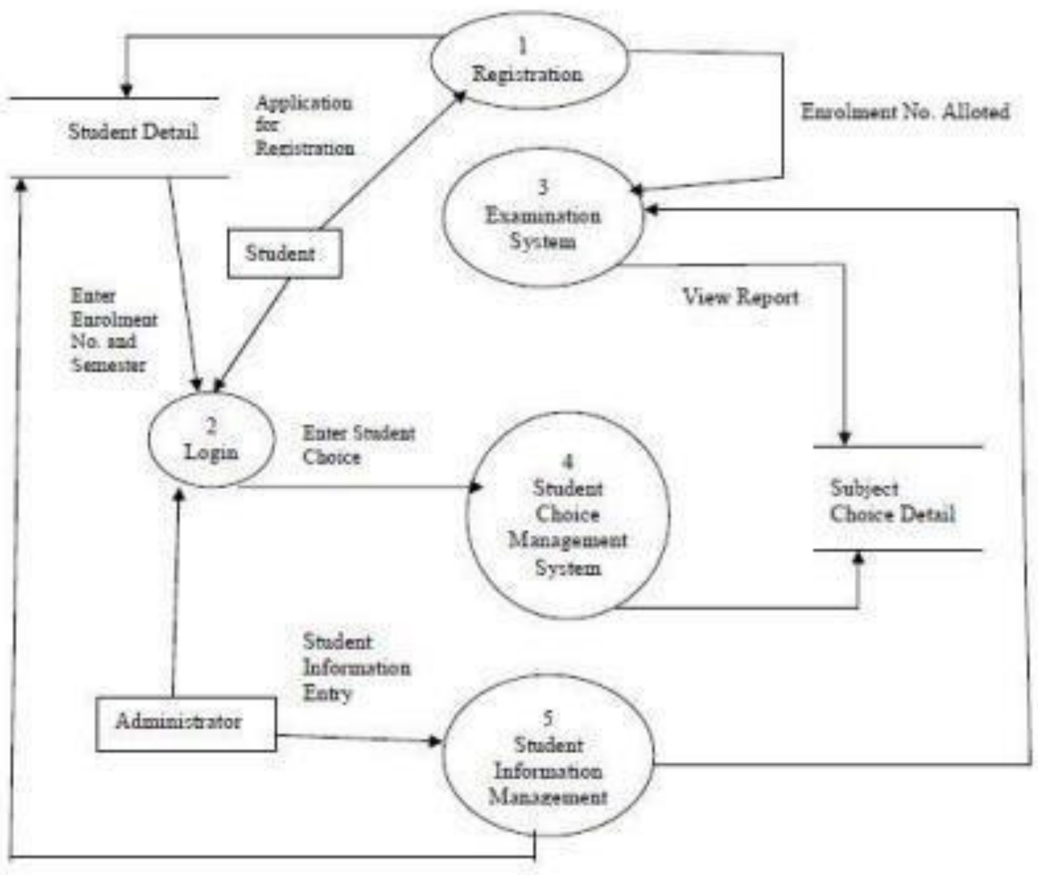

1
Registration
Student Detail
Application for Registration
Enrolment No. Alloted
3
Examination System
Student
View Report
Enter Enrolment No. and Semester
2
Login
Enter Student Choice
4
Student Choice Management System
Subject Choice Detail
Student Information Entry
Administrator
5
Student Information Management

Fig 4.8 Level 1 DFD of Student Admission and Examination System

Level 2 DFD

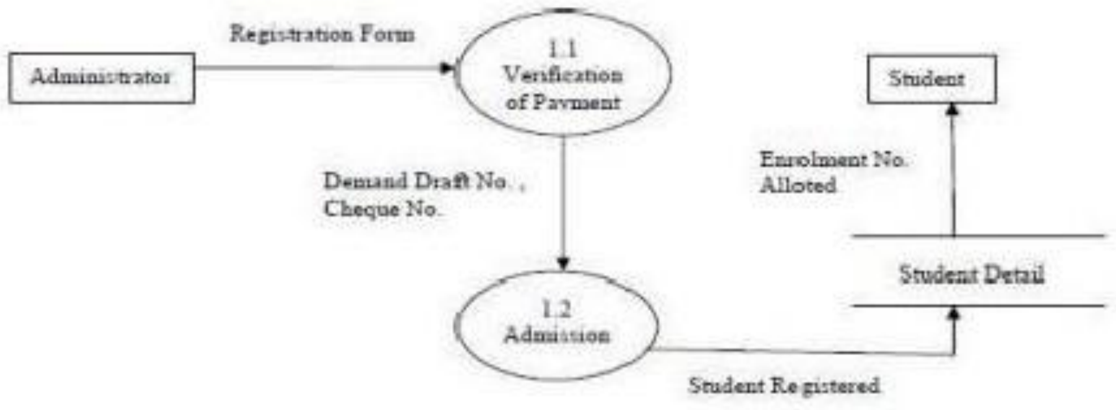


Fig.4.9 Level 2 DFD of Registration

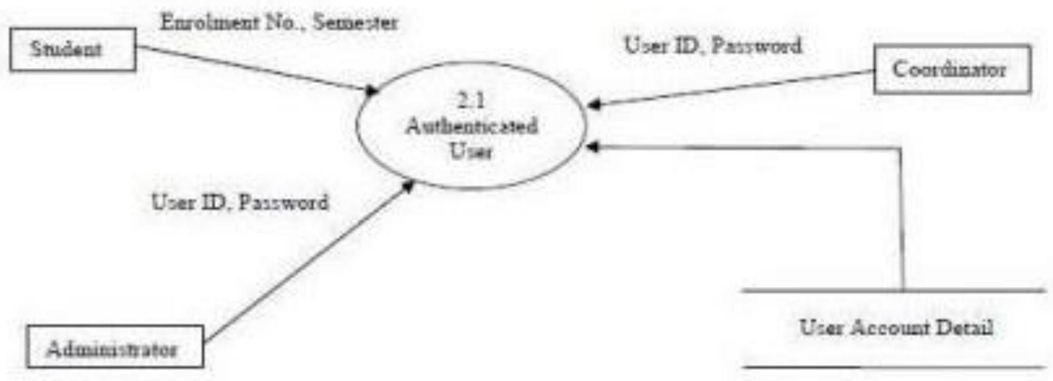


Fig. 4.10 Level 2 DFD of Authenticated

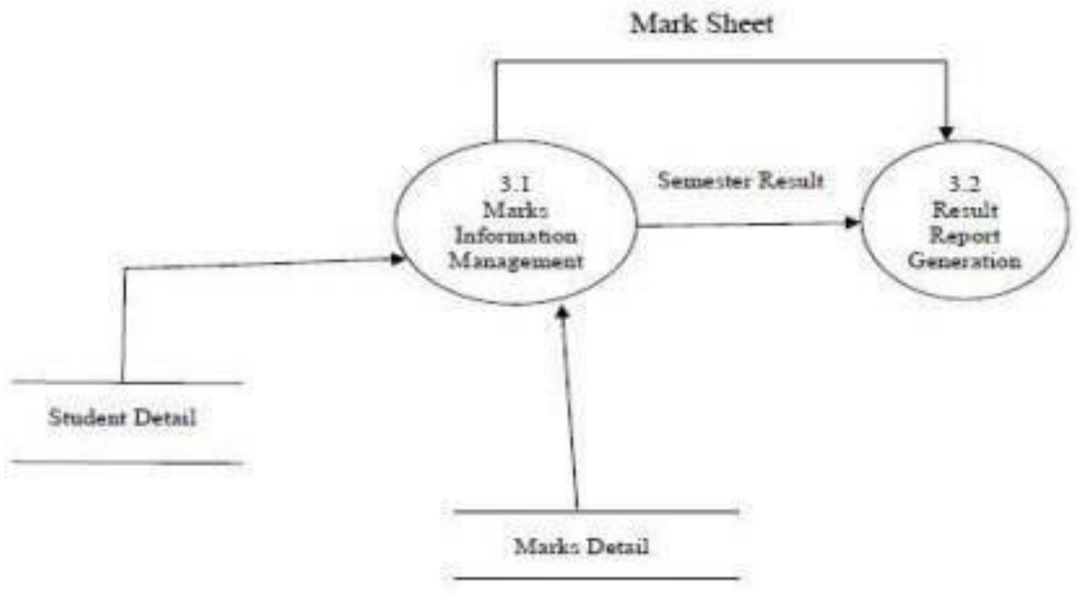

Mark Sheet
3.1 Marks Information Management
Semester Result
3.2 Result Report Generation
Student Detail
Marks Detail

Fig 4.11Level 2 DFD of Examination

Shortcomings 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 level and 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 that is a DFD representation into a structured chart.
- A structured chart represents the software architecture i.e. The various modules making up the system, the module dependency and the parameters that are passed among the different modules.
- The structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on module structure of a software and the interaction among the different modules.
- The procedural aspects are not represented in a structured design. The basic building blocks which are used to design structure charts are:

Rectangular boxes: A rectangular box represent module

Module invocation arrows: An arrow connecting two modules implies that during program execution, control is passed from one module to the other in the direction of the connecting arrow.

Data flow arrows: These are small arrows appearing alongside the module invocation arrows.The data flow arrows are annotated with the corresponding data name. The data flow arrows represents the fact that the named data passes from one module to the other in the direction of the arrow.

Flow Chart vs Structure Chart:

A flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:

➢ It is usually difficult to identify different modules of the software from its flow chart representation.

➢ Data interchange among different modules is not represented in a flow chart.

➢ Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

Principles of transformation of DFD to Structure Chart

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

➢ Transform analysis

➢ Transaction analysis

Normally, one starts with the level 1 DFD, transforms in into module representation using either the transform or the transaction analysis and then proceeds towards the lower-level DFDs. At each level of transformation, first determine whether the transform or the transaction analysis is applicable to a particular DFD.

Transform analysis and Transaction Analysis

Transform Analysis

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

➢ Input

➢ Logical processing

➢ Output

- The input portion in the DFD includes processes that transform input data from physical to logical form. Each input portion is called an afferent branch.
- The output portion of a DFD transforms output data from logical form to physical form. Each output portion is called an efferent branch. The remaining portion of a DFD is called central transform.
- In the next step of transform analysis, the structure chart is derived by drawing one functional component for the central transform and the afferent and efferent branches. Identifying the highest level input and output transforms require experience and skill.

Transaction Analysis

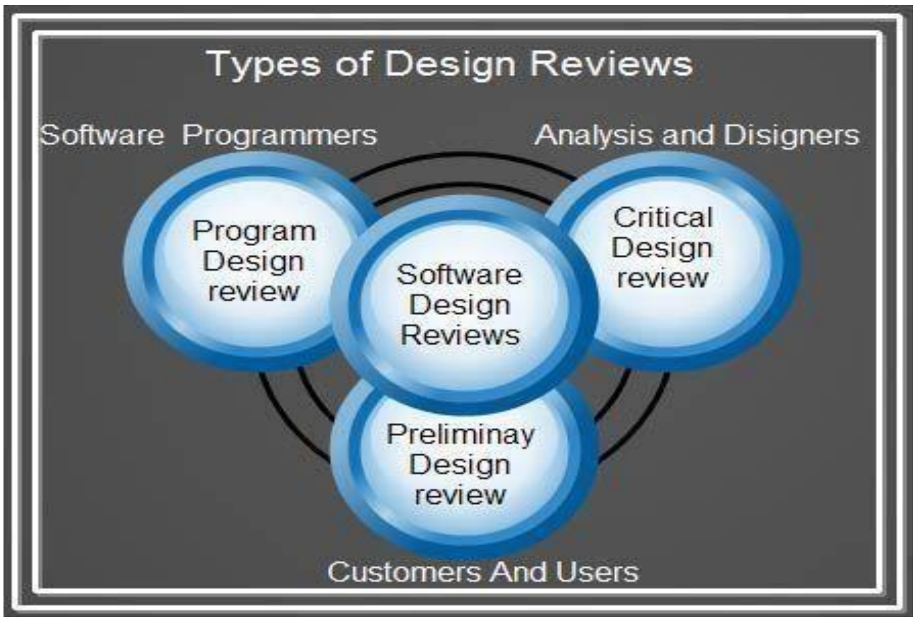
- A transaction allows the user to perform some meaningful piece of work.
- In a transaction-driven system, one of several possible paths through the DFD is traversed depending upon the input data item.
- Each different way in which input data is handled in a transaction. The number of bubbles on which the input data to the DFD are incident defines the number of transactions.
- Some transactions may not require any input data.
- For each identified transaction, we trace the input data to the output. In the structure chart, we draw a root module and below this module we draw each identified transaction of a module.

Design Review

Types of Software Design Reviews

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

- **preliminary design review** is conducted with the customers and users to ensure that the conceptual design (which gives an idea to the user of what the system will look like) satisfies their requirements.
- **critical design review** is conducted with analysts and other developers to check the technical design (which is used by the developers to specify how the system will work) in order to critically evaluate technical merits of the design.
- **program design review** is conducted with the programmers in order to get feedback before the design is implemented.



CHAPTER5
USER INTERFACE DESIGN

Articles to covered
Characteristics of Good Interface
Basic concepts of UID
Types of User interfaces
Components based GUI development

Characteristics of good user interface

1. Speedoflearning

A good user interface design is easy to learn. The learning speed is just progressed by using complex syntaxes and semantics of the command issue procedures. There is no need to learn the commands by users in a good user interface. A good user interface also does not allow its user to remember information of different screens while doing any task.

2. Attractiveness

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

3. Consistency

- The user interface should have a more consistency.
- Consistency also prevents online designers information chaos, ambiguity and instability.
- We should apply typeface, style and size convention in a consistent manner to all screen components that will add screen learning and improve screen readability

4. Feedback

Providing remarks to the moves of the person facilitates person to apprehend processing of the system. If any request of user takes more than a few seconds then user starts to panic, that is what is happening, if the proper feedback is providing to user, then he must know about his actions. Thus, a good user interface must contain feedback about the processing.

5. Error recovery

Error could be very common, all people can dedicate an blunders even specialists also can dedicate mistakes. Therefore, it’s also a responsibility of a great person interface to offer a undo facility in order that person can get better their errors at the same time as use of the interface. If the mistakes can't be recovered through users, they experience irritated, helpless, and low**.**

Basic concepts of UID

The graphical user interface, developed in the late 1970s by the Xerox Palo Alto research laboratory and deployed commercially in Apple's Macintosh and Microsoft's Windows operating systems, was designed as a response to the problem of inefficient usability in early, text-based command-line interfaces for the average user.

Graphical user interface design principles conform to the model–view–controller software pattern, which separates internal representations of information from the manner in which information is presented to the user, resulting in a platform where users are shown which functions are possible rather than requiring the input of command codes.

Types of User interfaces

User interfaces can be classified into the following three categories:

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

1. Command language based interfaces

✓ A command language-based interface as the name itself suggests, is based on designing a command language which the user can use to issue the commands.

✓ The user is expected to frame the appropriate commands in the language and type them in appropriately whenever required.

✓ A simple command language-based interface might simply assign unique names to the different commands. However, a more sophisticated command language-based interface may allow users to compose complex commands by using a set of primitive commands.

✓ Command language-based interfaces allow fast interaction with the computer and simplify the input of complex commands.

2. Menu-based interfaces

✓ An important advantage of a menu-based interface over a command language-based interface is that a menu-based interface does not require the users to remember the exact syntax of the commands.

✓ A menu-based interface is based on recognition of the command names, rather than recollection.

✓ Further, in a menu-based interface the typing effort is minimal as most interactions are carried out through menu selections using a pointing device. This factor is an important consideration for the occasional user who cannot type fast.

✓ menu-based user interface to be slower than a command language-based interface.

3. Direct manipulation interfaces

✓ Direct manipulation interfaces present the interface to the user in the form of visual models (i.e. icons or objects). For this reason, direct manipulation interfaces are sometimes called as iconic interface.

✓ In this type of interface, the user issues commands by performing actions on the visual representations of the objects, e.g. pull an icon representing a file into an icon representing a trash box, for deleting the file.

✓ Important advantages of iconic interfaces include the fact that the icons can be recognized by the users very easily, and that icons are language-independent.

✓ However, direct manipulation interfaces can be considered slow for experienced users.

✓ Also, it is difficult to give complex commands using a direct manipulation interface. For example, if one has to drag an icon representing the file to a trash box icon for deleting a file, then in order to delete all the files in the directory one has to perform this operation.

Components based GUI development

✓ A development style based on widgets (window objects) is called component-based (or widget-based) GUI development style.

✓ There are several important advantages of using a widget-based design style. One of the most important reasons to use widgets as building blocks is because they help users learn an interface fast.

CHAPTER 6
Software Coding & Testing

Articles to be covered

6.1 Coding

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

Representative coding standards are:

1 . Rules for limiting the use of global:

These rules list what types of data can be declared global and what cannot.

2. Contents of the headers preceding codes for different modules:

The information contained in the headers of different modules should be standard for an organization. The exact format in which the header information is organized can also be specified.

Some standard header data are:
a) Name of the module
b) Date on which the module was created
c) Author's name
d) Modification history

e) Synopsis of the module
f) Different functions supported along with their input/output parameters
g)Global variables accessed / modified by the modules

3. Naming conventions for global variables, local variables and constants identifiers:

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

4. Error return conventions and exception handling mechanisms:

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

Code Review

Code walk through

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

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

❖ Each member selects some test cases and simulates execution of the code through differentstatements 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.

Code inspections and software Documentation

Code inspections

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 (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:

o Internal documentation
o 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.

Unit Testing

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

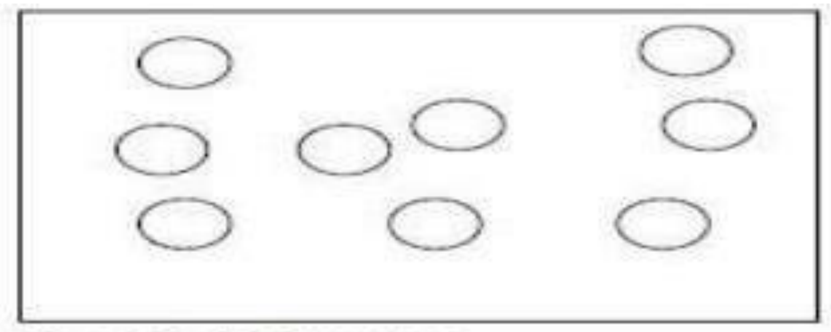


Fig. 6.1 Unit testing

- ❖ 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:

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

eliminated.

Methods of Black –Box Testing

In the black-box testing, test cases are design from an examination of the input/output values only and no knowledge of design or code is required.

Two main approaches to design black-box test cases are:

● Equivalence class Partitioning
● Boundary value analysis

Equivalence class partitioning and boundary value analysis **Equivalence**

Class Partitioning

❖ In the equivalence class partitioning approach, the domain of input values to a program is partitioned into a set of equivalence classes.

❖ The partitioning is done such that the behavior of the program is similar to 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 both the input and output data

❖ Example – Suppose we have to develop a software that can calculate the square root of an input integer . The value of the integer lies between 0 and 5000. As the input domain of such software is 0 to 5000,

so the equivalence class Of the software will be 0 to 5000 .This equivalence class can be partitioned into the following three equivalence classes 1. equivalence classes 1-The input integers whose value is less then 0.(invalid)
2. equivalence classes 2-The input integers whose value lies between 0 and 5000.(valid)
3. equivalence classes 3- The input integers whose value is greater than 5000.(invalid).

Boundary Value Analysis

● Boundary Value Analysis concentrates on the behavior of the system on its boundaries of its input variables.
● The boundary of a variable includes the maximum and the minimum valid value it is allowed attain in the system.

● It may be an input or output or even some internal future or variable of the system that captures some information of the system.
● Behavior of the system at its boundaries is tested under boundary value analysis.
● Boundary value analysis-based test suite design involves designing test cases using the values at the boundary of different equivalence classes.

EX:-For the above software that calculates the square root of integer values in the range between 0 and 5000 the test case can be designed as follows i.e.

{0,-1,5000,5001}

White Box Testing

• White –Box testing is also known as transparent testing.
• It is a test case design method that uses the control structure of the procedural design to derive test cases.
• It the most widely utilized unit testing to determine all possible path with in a module, to execute all looks and to test all logical expressions.
• This form of testing concentrate on procedural detail.

The general outline of the white-box testing process is:

• Perform risk analysis to guide entire testing process.
• Develop a detailed test plan that organizes the subsequence testing process. •
Prepare the test environment for test execution.
• Execute test cases and communicate the results.
• Prepare a report

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

1 . 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.

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 return x;

}
```

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)}.

2. Branch Coverage

In the branch coverage-based testing strategy, test cases are designed to make each branch condition assume true and false value in turn.

Brach testing is also known as edge testing, which is stronger than statement coverage 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.

Test case1 x=6,y=6
Test case2 x=6,y=7
Test case3 x=8,y=7
Test case4 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)}.

3. 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 ((C1 AND C2) OR C3), the components C1,C2 andC3 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.

4. Path Coverage

- The path coverage-based testing strategy requires designing test cases such that all linearly independent paths is the program are executed at least once.

- A linearly independent path can be defined in the terms of the control flow graph (CFG) of a program.

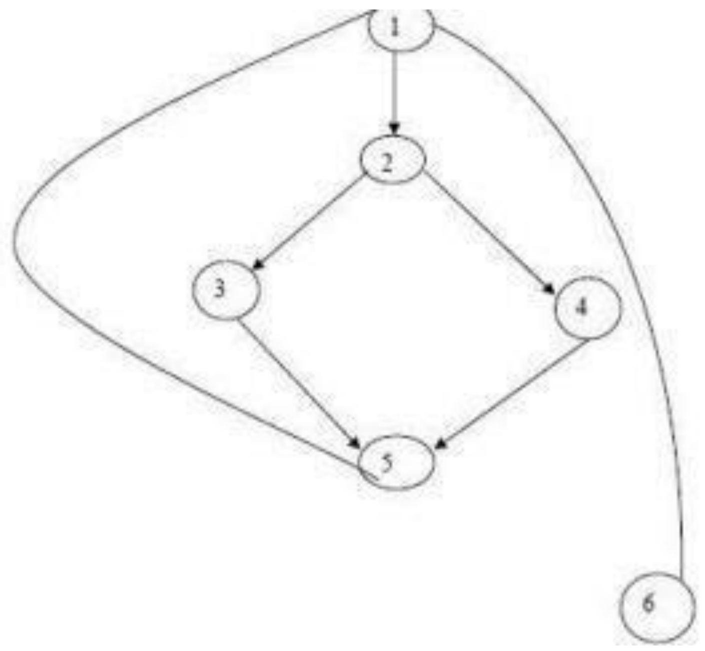
Control Flow Graph (CFG)

- A control flow graph describes the sequence in which the different instructions of a program get executed.

- The flow graph is a directed graph in which nodes are either entire statement or fragments of a statement and edges represents flow of control.

- An edge from one node to another exists if the execution of the statement representing the first node can result in the transfer of control to the other node.

- A flow graph can easily be generated from the code of any problem.



int computer_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;

}

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

- Cyclomatic complexity defines an upper bound on the number of independent paths in a program.
- Given a control flow graph G of a program. Each node of the graph represents a command or a statement of the program and each edge represents the flow of execution between statements or nodes.
- For a control flow graph with E number of edges and N number of nodes, the

cyclomatic complexity can be computed as

M = E – N + 2P

Where P is the number of connected components in the graph.

- Control flow graph of a sequential program is a single component graph, Hence, for any sequential program M = E – N + 2

Example:

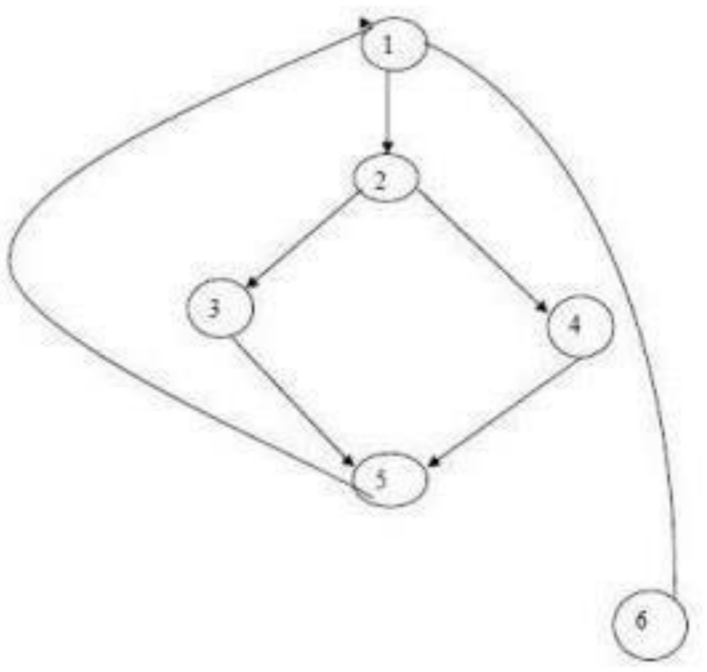


Fig. 6.4 Control Flow Graph

Number of Edges = E = 7

Number of Nodes = N = 6

The value of cyclomatic complexity is

V(G) = E – N + 2

= 7 – 6 + 2

= 3

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.

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 the mutant.
- 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 approaches

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

a. 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.

b. 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.

c. 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.

d. 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.

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 partial integrated system is tested.

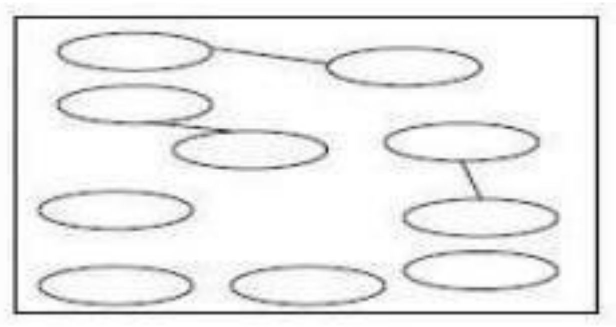


Fig.6.5 Integration Testing

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

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

1. 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.

2. Top – Down Approach

• Top – down integration proceeds down the invocation hierarchy, adding are module at a time until an entire tree level is integrated and it elements the need for drivers. • In this approach testing can start only after the top-level modules have been coded and unit tested.

• 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 lower– level routines, many times it may become difficult to exercise the top- level routines in the desired manner since the lower – level routines perform several low level functions such I/O.

3. 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 communicated among each other through well– defined interfaces.

• The primary purpose of testing each subsystem is to test the interface among various modules making up the subsystem.

• Both control and data interfaces are tested. Advantages of bottom – up integration testing is that several disjoint subsystems can be tested simultaneously.

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

4. Mixed Integration Testing

A mixed(also called sandwiched) integration testing follows a combination of top – down and bottom – up testing approaches. In this approach testing can start as and when modules become available.

Phased and Incremental integration 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.

Phased integration requires less number of integration steps compared to the incremental integration approach. However, when failures are detected, it is easier to debug the system in the incremental testing approach since it is known that the error is caused by addition of a single module. In fact, big bang testing is a degenerate case of the phased integration testing approach

System testing alphas beta and acceptance testing

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

- ➢ Alpha testing
- ➢ Beta testing
- ➢ Acceptance testing

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 performance test 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 non functional requirements of the system.

Performance Testing, Error seeding

- ➢ Performance testing is carried out to check whether the system meets the non functional requirements identified in the SRS document.
- ➢ The types of performance testing to be carried out on a system depend on the different Non functional requirements of the system document in the SRS document
- ➢ All performance tests can be considered as black – box tests.

Error Seeding

- ➢ Error seed can be used to estimate the number of residual errors in a system.
- ➢ Error seeding seeds the code with some known errors.
- ➢ The number of seeded error detected in the course of standard testing procedure determined. These values in-conjunction with the number of unseeded errors can be used to predict:
  - i) The number of errors remaining in the product
  - ii) The effectiveness of the testing method
- ➢ Let n be the total number of errors in the system and let "n" number of these errors are detected during testing.
- ➢ Let "S" be the total number of seeded errors and let "s" be the number of these errors are detected during testing.

  n /N = s/ S

  => N = S * n /s

  => (N-n) = n(S-s) /S

General issues associated with testing

Some general issues associated with testing

i)Test documentation

ii) Regression testing

Test Documentation

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

Regression Testing

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

Chapter 7
Software Reliability

Articles to be covered

Software Reliability
Different reliability metrics
Reliability growth modeling
Software quality
Software Quality Management System

Software Reliability

✓ Reliability of a software product can be defined as the probability of the product working correctly over a given period of time.

✓ A software product having a large number of defects is unreliable. Reliability of a system improves it the number of defects in it is reduced.

✓ The reliability of a product depends on the both the number of errors and the exact location of the errors. Reliability also depends upon how the product is used (i.e. on its execution profile).

✓ Different users use a software product in different ways. So defects which show up for one user may not show up for another user.

Different reliability metrics

Some reliability metrics which can be used to quantity the reliability of software products are:

1. Rate of Occurrence of Failure (ROCOF)

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

2.Probability of Failure ON Demand (POFOD)

POFOD measures the likelihood of the system failure when a service request is made.

For example a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure.

Availability

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

In order to intimately, it is necessary to classify various types of failures. Possible classifications of failures are:

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

3. 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.

4. Mean Time to Repair (MTTR)

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

Mean Time Between Failures (MTBF)

MTBF = MTTF+MTTR

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

Reliability growth modelling

✓ 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 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. Two very simple reliability growth models are :

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.

✓ However this simple model of reliability which implicitly assumes that all errors contribute equally to reliability growth, is highly unrealistic.

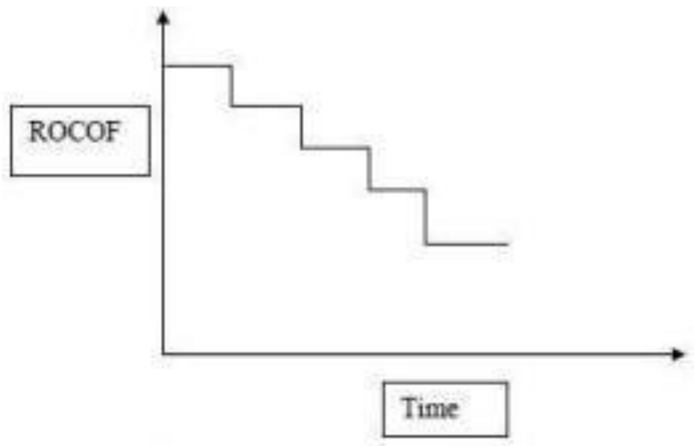


Fig.7.1 Step function model of reliability growth

Software quality

The objective of software engineering is to produce good quality maintainable software in time and within budget. That is, a quality product does exactly what the users want it to do.

The modern view of quality associates a software product with several quality factors such as :

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

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

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

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

Maintainability: A software product is maintainability, 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 easily modified, etc.

Software Quality Management System

Software Quality Management System

Issues associated with a quality system are:

Management structural and individual responsibilities A quality system is actually the responsibility of the organization as a whole.

However, many organization have a separate quality department to perform several quality system activities. The quality system of an organization should have the support of the top management

Quality system activities

- ✓ Auditing of the 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.

A good quality system must be well documented.

Evolution of Quality Systems Quality system have rapidly evolved over the last 5 decades. The quality systems of organisation have undergone through 4-stages of evolution as :

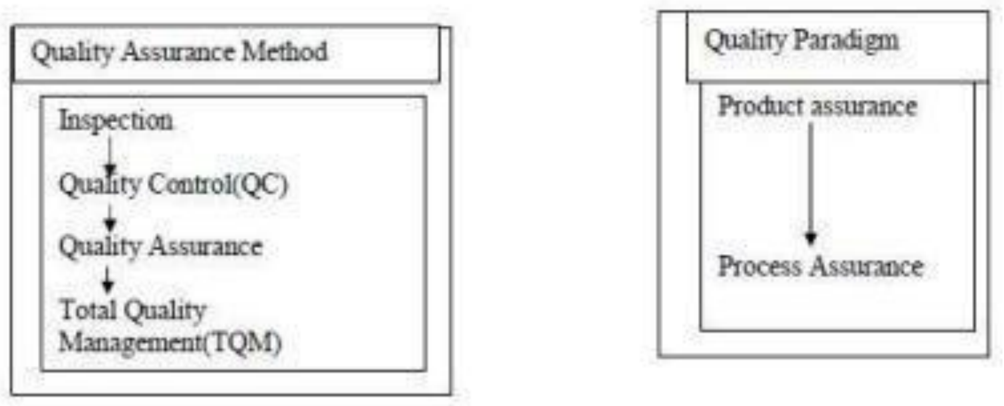


Fig. 7.2Evolution of quality system and the corresponding shift in the quality paradigm.

- ✓ Quality control focuses not only on detecting the defective product & eliminating them. But also on determining the causes behind the defects.
- ✓ The quality control aims at correcting the causes of errors & not just rejecting the defective products.
- ✓ The basic premises of modern quality assurance is that if an organizations processes are good and are followed rigorously then the products are bound to be of good quality.
- ✓ The modern quality paradigm includes some guidance for recognising, defining, analysing & improving the production process.
- ✓ Total quality management (TQM) says that the process followed by an organisation must be continuously improve through process measurement.