EDUSAT LEARNING RESOURCE MATERIAL

On

Software Engineering

(For 5th Semester CSE & IT)

Prepared by

1. **Er. Ramesh Chandra Sahoo, Sr. Lect (CSE & IT) ,**
   UCP Engg. School, Berhampur
2. **Miss Sasmita Panigrahi, PTGF(CSE & IT),**
   UCP Engg. School, Berhampur

Course Contents

Chapter - 1

Introduction to Software Engineering

Contents

1.1 Relevance of Software Engineering

Software engineering is the field of computer science that deals with the building of software systems which are so large or so complex that they are build by a team or teams of engineers.

Parnas has defined software engineering as “multi-person construction of multi-version software”.

According to Fritz Bauer, software engineering is “The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines”.

Stephen Schach defined as “ A discipline whose aim is the production of quality software, software that is delivered on time, within budget, and that satisfies its requirements”.
Software has become critical to advancement in almost all areas of human endeavour. The art of programming only is no longer sufficient to construct

large programs. There are serious problems in the cost, timeliness, maintenance and quality of many software products.

The foundation for software engineering lies in the good working knowledge of computer science theory and practice. The theoretical background involves knowing how and when to use data structures, algorithms and understanding what problems can be solved and what cannot. The practical knowledge includes through understanding of the workings of the hardware as well as thorough knowledge of the available programming languages and tools.

Software engineering has the objective of solving these problems by producing good quality, maintainable software, on time, within budget. To achieve this objective, we have to focus in a disciplined manner on both the quality of the product and on the process used to develop the product.

1.2 Software Characteristics and Applications

Software is a logical rather than a physical system element. Its characteristics that make it different from other things human being build.

- Software is developed or engineered, it is not manufactured in the classical sense which has quality problem.
- Software does not “wear out”, but it deteriorates due to change.
- Although the industry is moving toward component-based construction, most software continues to be custom-built. Modern reusable components encapsulate data and processing into software parts to be reused by different programs. E.g. graphical user interface, window, pull-down menus in library etc.

Software Applications

Software may be applied in any situation for which a pre-specified set of procedural steps has been defined. Information content and determinacy are import factors in determining the nature of a software application. Contents refer to the meaning and form of incoming and outgoing information. Applications are:

- System software: System software is a collection of programs written to service other programs. Examples of system software are compilers, editors, file management utilities, operating system components, drivers.
- Application software: Stand-alone programs for specific needs.
- Engineering / scientific software: Characterized by “number crunching” algorithms. Such as automotive stress analysis, molecular biology, orbital dynamics etc.
- Embedded software resides within a product or system.
- Product-line software focus on a limited marketplace to address mass consumer market.
- Web based software, the web pages retrieved by a browser are software that incorporates executable instructions and data. As web 2.0 emerges, more sophisticated computing environments is supported integrated with remote database and business applications.
- AI software uses non-numerical algorithm to solve complex problem. Examples are Robotics, expert system, pattern recognition, game playing.

1.3 Emergence of Software Engineering

Software engineering techniques have evolved over many years which resulted 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 briefly describe in Article 1.4.

1.4 Early Computer Programming, High Level Language Programming, Control Flow Based Design, Data Flow Oriented Design, Data Structure Oriented Design, Object and Component Bases Design

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 writing the programs in his own style.

High-Level Language Programming

Computers become faster with the introduction of the 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 programs instructions are executed.

Data Structure-Oriented Design

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

Object-Oriented Design

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

1.5 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.

1.6 Classical Waterfall Model and Iterative Waterfall Model

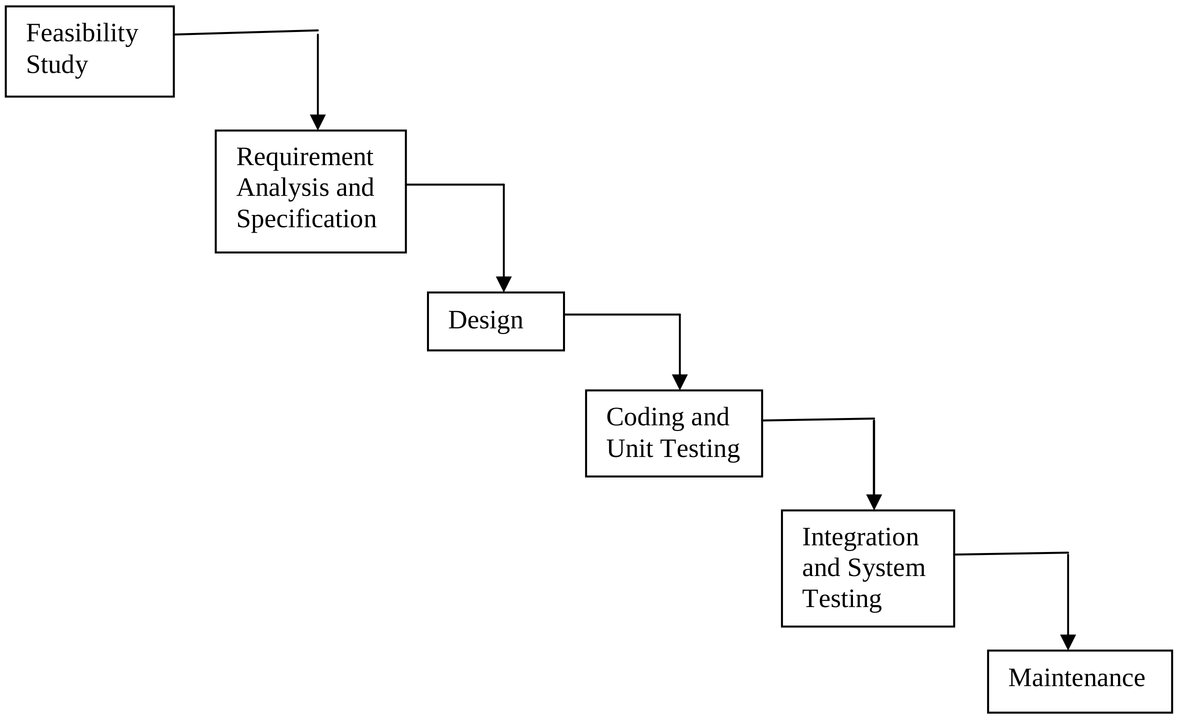
This model is called as linear sequential model. This model suggests a systematic approach to software development.

The project development is divided into sequence of well-defined phases. It can be applied for long-term project and well understood product requirement.

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 sequence 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 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.

System testing is normally carried out in a planned manner according to a system test plan document. The results of information and system testing are documented in the form of a test report.

Maintenance

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

- **Corrective Maintenance**

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

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

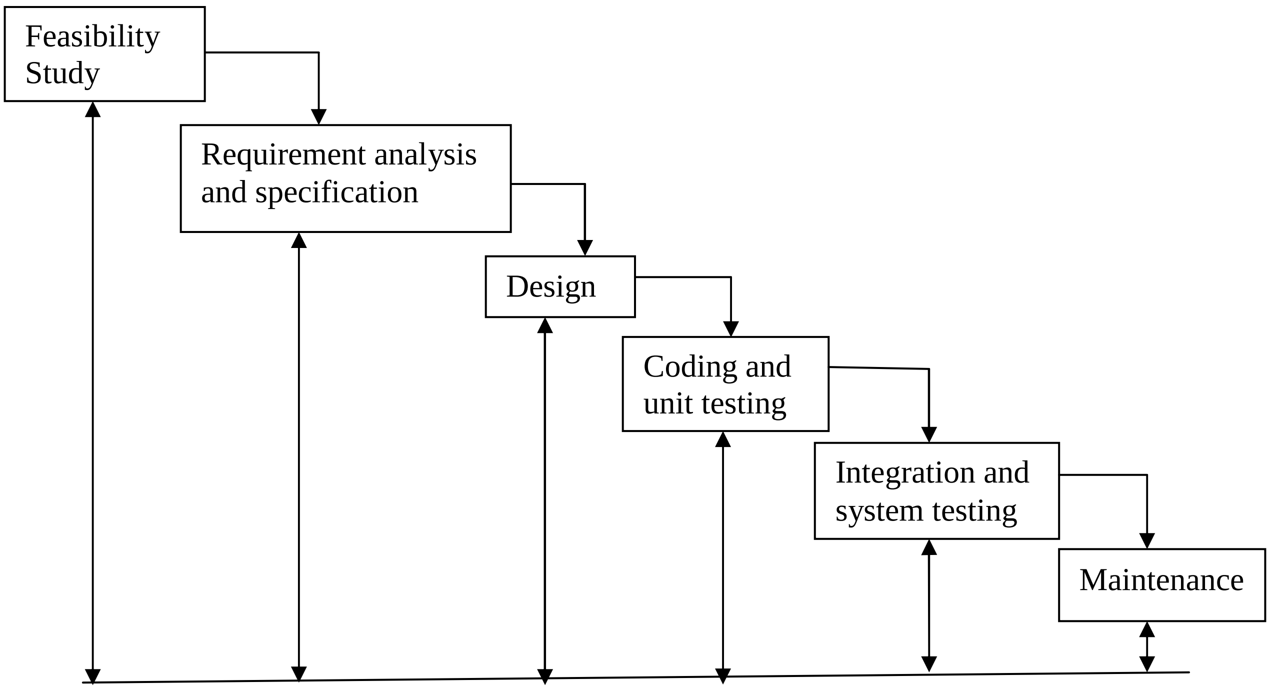
  Adaptive maintenance is usually required for reporting the softer to work in a new environment.

Iterative Waterfall Model

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. However in practical development environment, the engineers do commit a large number of errors in different phases of the life cycle. The source of the defects can be wrong assumptions, use of in appropriate technology, communication gap among the project developers etc. These defects usually get detected much later in the life cycle. 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.



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

Fig. 1.2 Iterative waterfall Model

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 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.

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.

1.7 Prototyping Model

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

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

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

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 development.

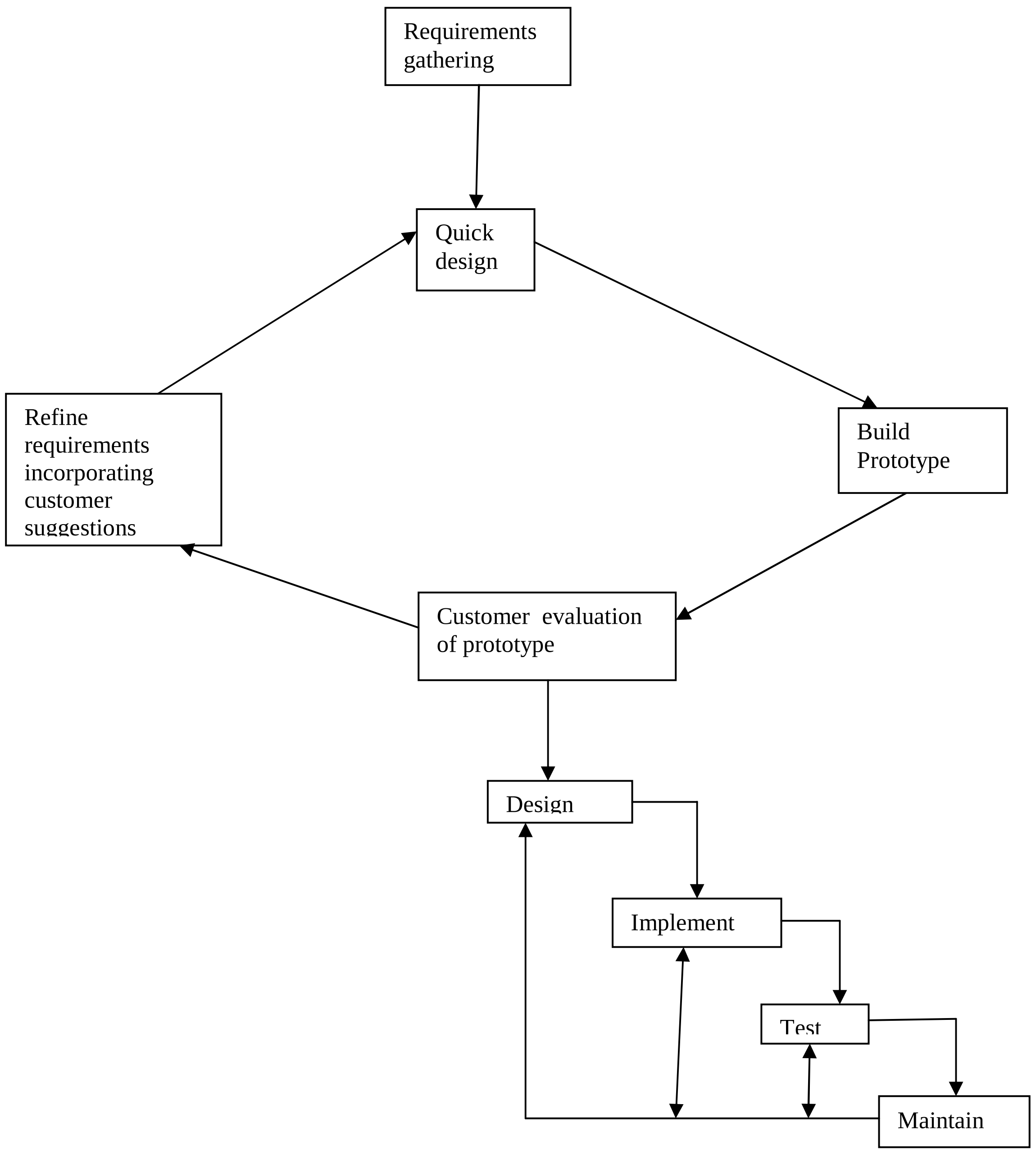

Requirements gathering
Quick design
Build Prototype
Customer evaluation of prototype
Refine requirements incorporating customer suggestions
Design
Implement
Test
Maintain

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 feedback to the developers regarding the prototype: what is correct, what needs to be modified, what is missing, what is not needed etc. Based on the customer feedback the prototype is modified and then the users and the clients are again allowed to use the system. This cycle of obtaining customer feedback and modifying the prototype continues till the customer approves the prototype.

After the finalization of software requirement and specification (SRS) document, the prototype is discarded and actual system is then developed using the iterative waterfall approach.

Prototyping is often not used, because that development costs may become large. However in some situations, the cost of software development without prototyping may be more than with prototyping. Prototype model is well suited for projects where requirements are hard to determine. This model requires extensive participation and involvement of the customer, which is not always possible.

1.8 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.

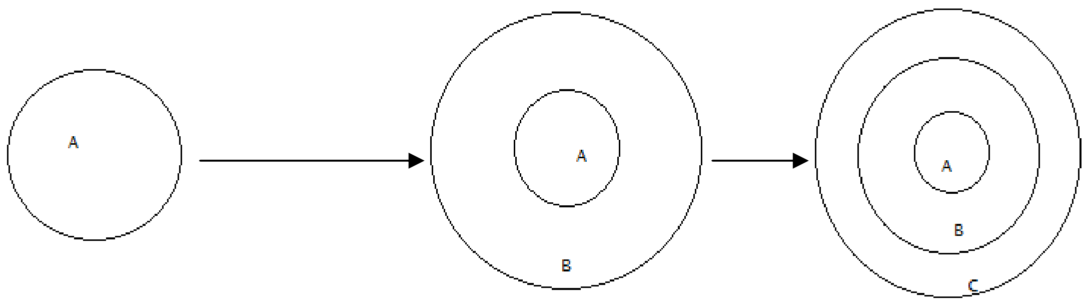
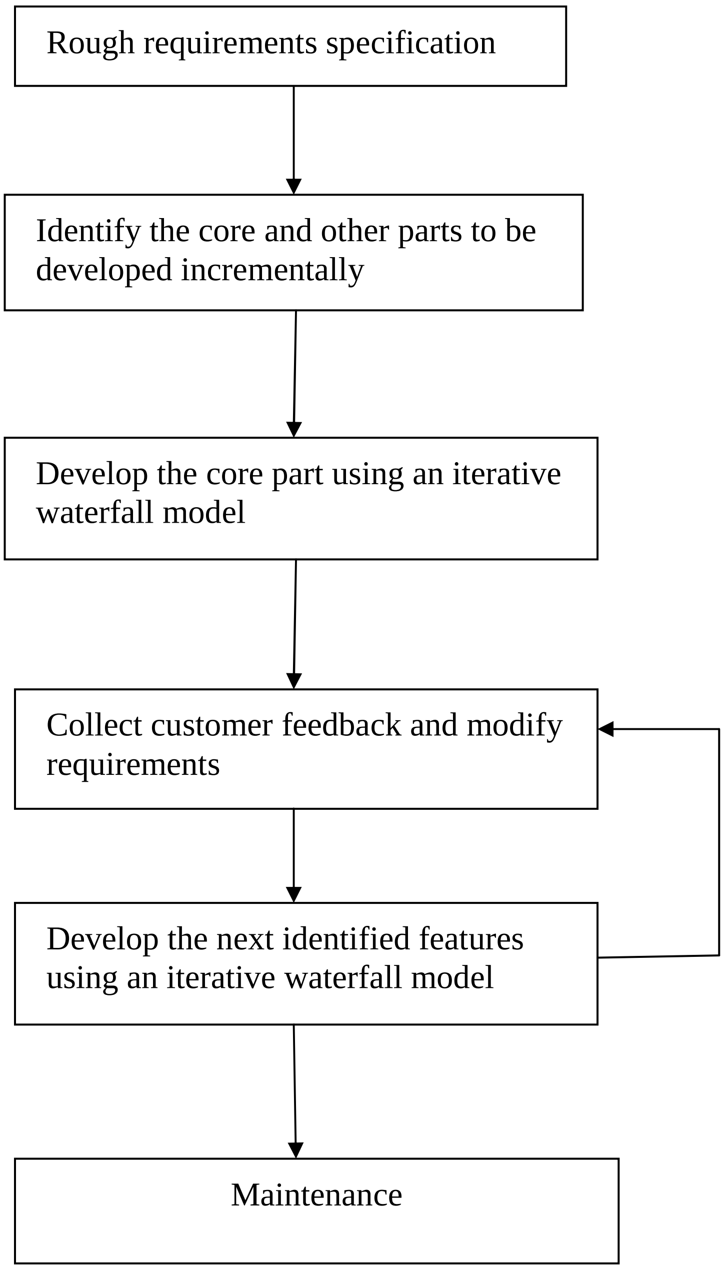


Fig. 1.4 Evolutionary model of software development

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.



Rough requirements specification
Identify the core and other parts to be developed incrementally
Develop the core part using an iterative waterfall model
Collect customer feedback and modify requirements
Develop the next identified features using an iterative waterfall model
Maintenance

Fig. 1.5 Evolutionary Model of Software Development

Each successive version of the product is fully functioning software capable of performing more useful work than the previous version. In this model the user gets a chance to experiment with partially developed software much before the complete version of the system is released. Therefore the evolutionary model helps to accurately elicit user requirements during the delivery of the different versions of the software and the change requests after delivery of the complete software are minimized.

The evolutionary model is used when the customer prefers to receive the products in increments rather than waiting for the full product to be developed and delivered. The evolutionary model is very popular for object oriented software development project.

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.

1.9 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.

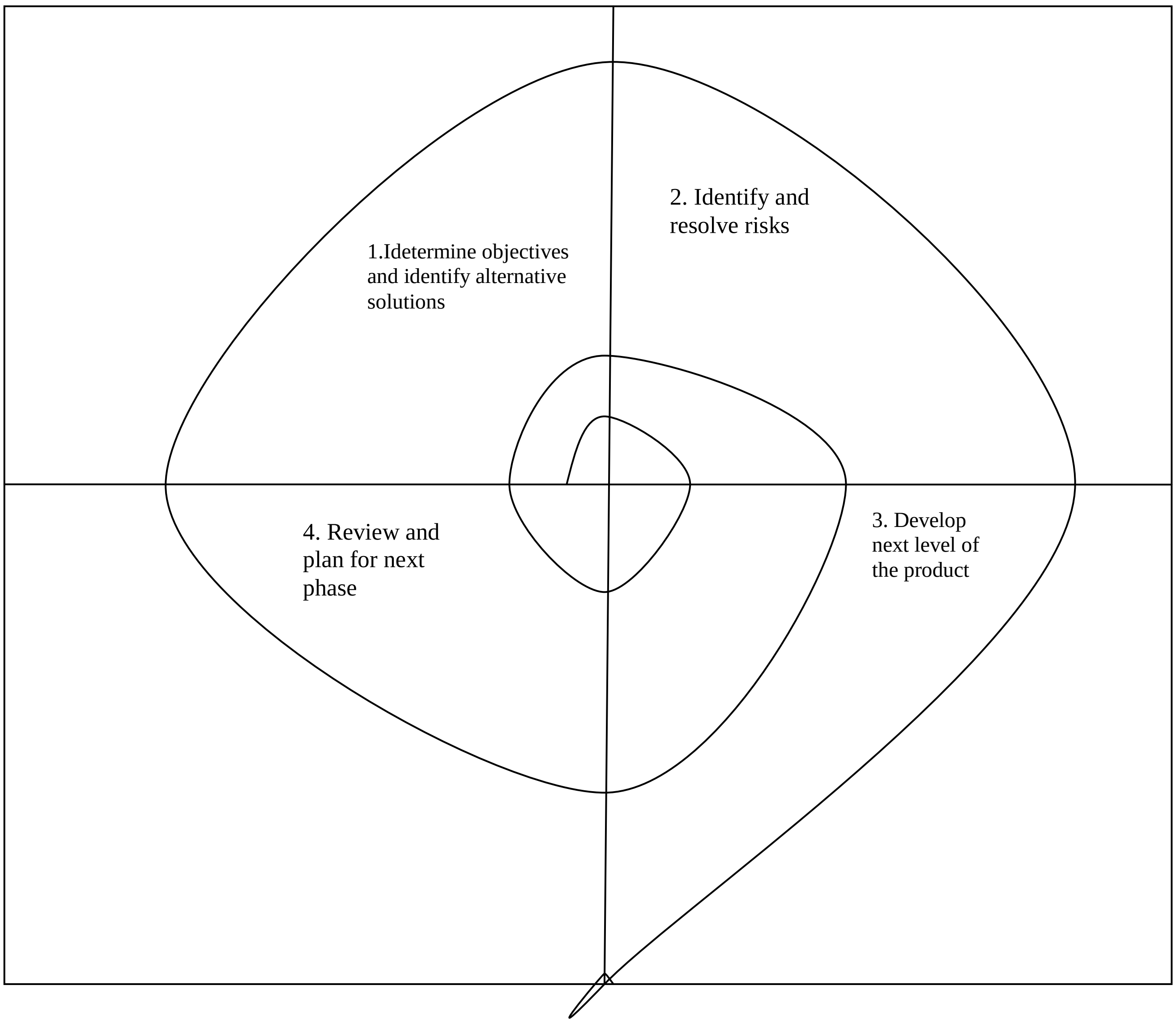

1.Idetermine objectives and identify alternative solutions
2. Identify and resolve risks
3. Develop next level of the product
4. Review and plan for next phase

Fig. 1.6 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:

- **Planning:** Identifies the objectives of the phase and the alternative solutions possible for the phase and constraints.
- **Risk analysis:** Analyze alternatives and attempts to identify and resolve the risks involved.

- **Development:** Product development and testing product.
- **Assessment:** Customer evaluation.

During the first phase planning is performed, risks are analyzed, 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
Understanding Project Management

Contents

2.1 **Project Management Concepts**

The main goal of software project management is to enable a group of software engineers to work efficiently towards successful completion of the project. The management of software development is dependent on four factors:

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

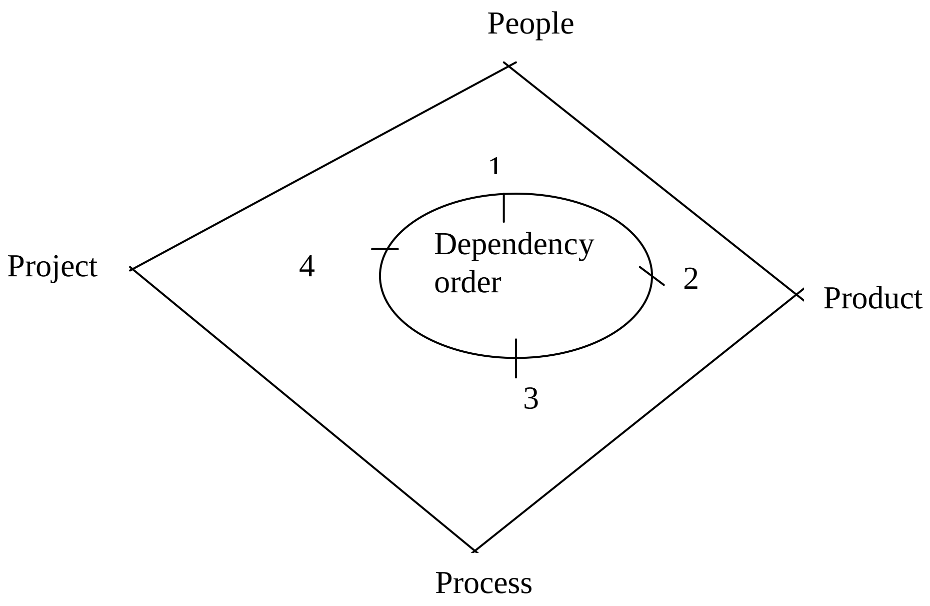


Fig. 2.1 Factors of Management Dependency

Effective software project management focuses on these items in this order:

- The people
  - Deals with the cultivation of motivated, highly skilled people.
  - Consists of the stack holders, the team leaders, and the software team.
- The Product
  - Product objectives and scope should be established before a project can be planned.
- The Process
  - The software process provides the framework from which a comprehensive plan for software development can be established.
- The Project
  - Planning and controlling a software project is done for one primary reason, it is the only known way to manage complexity.
  - In a 1998 survey, 26% of software projects failed outright, 46% experienced cost and schedule overruns.

2.2 Project Management

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.

Job Responsibilities of a Software Project Manager

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

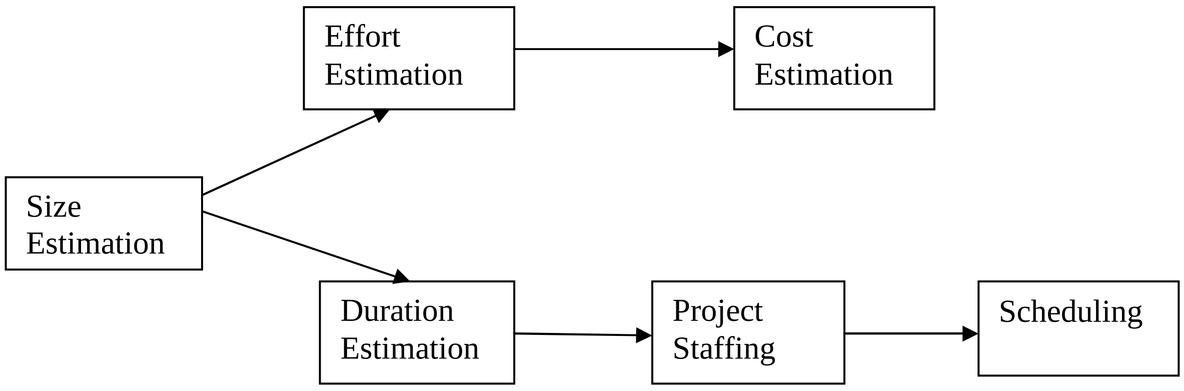


Fig. 2.2 Precedence Ordering among Planning Activities

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.

Sliding Window Technique

In this technique starting with an initial plan, the project is planned more accurately in successive development stages. At the start of a project, project manager have incomplete knowledge about the details of the project. The information gradually improves as the project progress through different phases. After the completion of every phase, the project manager can plan each subsequent phase more accurately and with increasing levels of confidence.

2.3 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 the source code occupies. The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.
Two metrics are widely used to estimate 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 measure correlates poorly with the quality and efficiency of the code. A larger code size does not necessary imply better quality or higher efficiency.
- LOC metrics penalizes use of higher level programming languages, code reuse etc.
- It is very difficult to accurately estimate LOC in the final product from the problem specification. The LOC count can be accurately computed only after the code has been fully developed.

Function Point Metric

- ♦ Function Points measure software size by quantifying the functionality provided to the user based solely on logical design and functional specifications
- ♦ Function point analysis is a method of quantifying the size and complexity of a software system 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 used for 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

Steps of Function Point Counting

- ♦ Determine the type of function point count
- ♦ Identify the counting scope and application boundary
- ♦ Determine the Unadjusted Function Point Count
- ♦ Count Data Functions
- ♦ Count Transactional Functions
- ♦ Determine the Value Adjustment Factor
- ♦ Calculate the Adjusted Function Point Count

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

The different parameters are:

- **Number Of Inputs:**
  Each data item input by the user is counted.
- **Number Of Outputs:**
  The outputs refers to reports printed, screen outputs, error messages produced etc.
- **Number Of Inquiries:**
  It is the number of distinct interactive queries which can be made by the users.
- **Number Of Files:**
  Each logical file is counted. A logical file means groups of logically related data. Thus logical files can be data structures or physical files.

- **Number Of Interfaces:**

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

  Function Point (FP) is estimated using the formula:

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

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

  TCF = DI (Degree of Influence) * 0.01

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

  Example- Once the unadjusted function point (UFP) is computed, the technical complexity factor (TCF) is computed next. The TCF refines the UFP measure by considering fourteen other factors such as high transaction rates, throughput and response time requirements etc. Each of these 14 factors is assigned a value from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). Now, the TCF is computed as (0.65+0.01*DI). As DI can vary from 0 to 70, the TCF can vary from 0.65 to 1.35.

  Finally FP = UFP *TCF

Feature Point Metric

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

Project Estimation Techniques

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

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

There are three broad categories of estimation techniques:

- Empirical estimation techniques
- Heuristic techniques
- Analytical estimation techniques

Empirical Estimation Techniques

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

Heuristic Techniques

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

- Single variable model
- Multivariable model

A single variable estimation model takes the following form:

$$\text{Estimated parameter} = c_1 * e^{d1}$$

Where e is a characteristics of the software, c_1 and d1 are constants.

A multivariable cost estimation model takes the following form:

Estimated Resource = $c_1 * e_1^{d1} + c_2 * e_2^{d2}$ + ...

Where e_1, e_2 ...are the basic characteristics of the software.

c_1, c_2, d1, d2 ...are constants.

Analytical Estimation Techniques

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

Halstead's Software Science an Analytical Estimation Techniques

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

For a given program, let:

- η_1 be the number of unique operators used in the program
- η_2 be the number of unique operands used in the program
- N_1 be the total number of operators used in the program
- N_2 be the total number of operands used in the program.

There is no general agreement among researchers on what is the most meaningful way to define the operators and operands for different programming languages.

For instance, assignment, arithmetic, and logical operators are usually counted as operators. A pair of parentheses, as well as a block begin and block end pair, are considered as single operators.

The constructs if......then.......else.....endif and a while......do are treated as single operators. A sequence operator ';' is treated as a single operator.

Operators and Operands for the ANSI C Language

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

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

Length and Vocabulary

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

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

Program Volume

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

$V = N \log_2 \eta$

The program volume V is the minimum number of bits needed to encode the program. In fact, to represent η different identifiers uniquely, we need at least $\log_2 \eta$ bits. We need $N \log_2 \eta$ bits to store a program of length N. 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. For any 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 and human resources. Hardware resources such as computer time, terminal time and memory required for the project, software resources include the tools and compilers needed 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, quality assurance, system integration, training. Bottom-up cost estimation first estimates the cost to develop each module or subsystem. Those costs are combined to arrive at an overall estimate. Two popular empirical estimation techniques are:

❖ 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 size after analyzing the problem thoroughly. The expert estimates the cost of the different modules or subsystems and then combines them to arrive at the overall estimate.

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

❖ Delphi Cost Estimation

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

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

2.5 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.
- **Semidetached:** In the semidetached mode the development team consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.
- **Embedded:** In the embedded mode of software development, the project has tight constraints, which might be related to the target processor and its interface with the associated hardware.

According to Boehm, software cost estimation should be done through three stages: basic COCOMO, intermediate COCOMO, and complete COCOMO.

Basic COCOMO

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

$$\text{Effort} = a_1 \times (\text{KLOC})^{a2} \text{ PM}$$

$$\text{Tdev} = b_1 \times (\text{Effort})^{b2} \text{ Months}$$

Where

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

(ii) $a_{1,}$ $a_{2,}$ $b_{1,}$ b_2 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 the size of the software and a set of other parameters known as cost directives. This model makes computations on the basis of 15 cost drivers based on the various attributes of software development. Cost drivers are used to adjust the nominal cost of a project to the actual project environment, hence increasing the accuracy of the estimate.
The cost drivers are grouped into four categories:

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

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

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

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

Development Environment

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

Complete COCOMO / Detailed COCOMO

Basic and intermediate COCOMO model considers a software product as a single homogeneous entity. Most large system are made up of several smaller subsystem. These subsystems may have widely different characteristics. Some subsystem may be considered organic type, some embedded and some semidetached. Software development is executed in different phases and hence the estimation of efforts and schedule of deliveries should be carried out phase wise. Detailed COCOMO provides estimated phase-wise efforts and duration of phase of development.

Detailed COCOMO classifies the organic, semidetached, and embedded project further into small, intermediate, medium and large-size projects based on the size of the software measured in KLOC. Based on this classification, the percentage of efforts and schedule have been allocated for different phase of the project, viz. software planning, requirement analysis, system designing, detailed designing, coding, unit testing, integration and system testing. Total effort is estimated separately. This approach reduces the margin of error in the final estimate.

2.6 Effect of Schedule Change on Cost

Only few number of engineers are needed at the beginning of the project to carry out planning and specification tasks. As the project progress and more detailed work is required, the number of engineers reaches a peak.

By using the Putnam's proposed expression for L,

$K = (L^3) / (C_k)^3 (t_d)^4$

Or

$K = C / (t_d)^4$ (Since $C = (L^3) / (C_k)^3$) is a constant)

- Where K is the total effort expended (in PM) in the product development and L is the product size in KLOC.
- t_d is the time required to develop the software.
- C_k is the state of technology constant and reflects constraints that impede the progress of the programmer.

From the expression, it can be observed that when the schedule of the project is compressed, the required effort increases.

The Putnam estimation model works reasonably well for very large systems, but seriously overestimates the effort on medium and small systems.

2.7 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:

$$L=C_{te}\, t_d\, K^{1/2}$$

Where C_{te} is the effective technology constant, t_d is the time to develop the software, and K is the effort needed to develop the software.

2.8 Tools for 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 to do 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 Review Technique chart representation is developed. The PERT chart representation is suitable for program monitoring and control.

2.9 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 undertake.

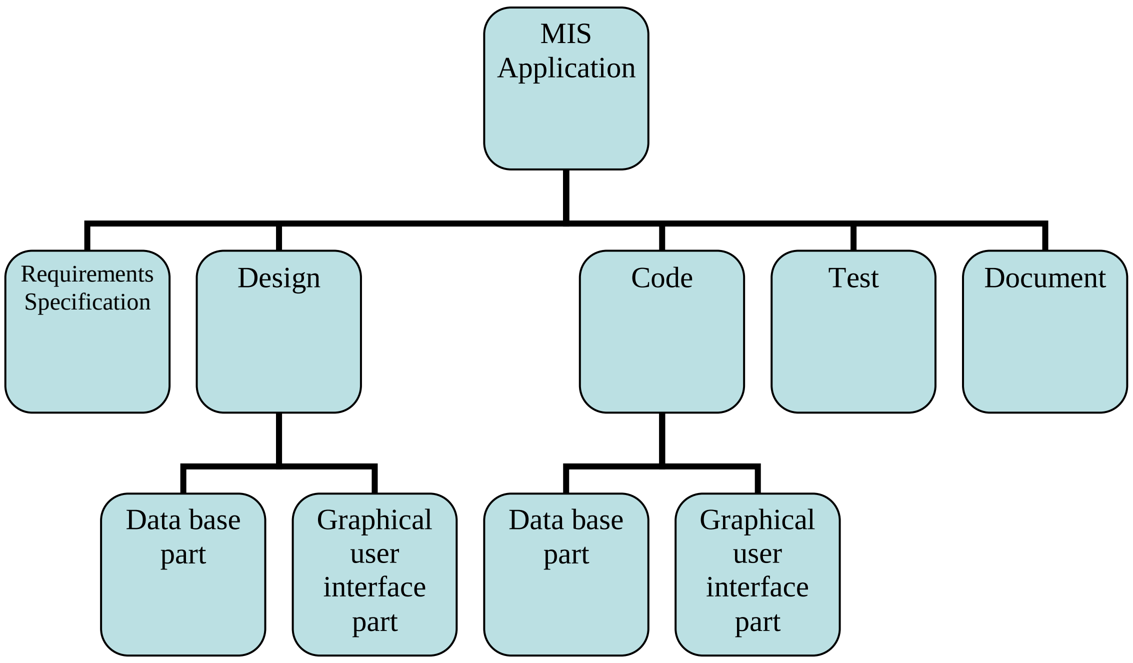

MIS Application
Requirements Specification
Design
Code
Test
Document
Data base part
Graphical user interface part
Data base part
Graphical user interface part

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.

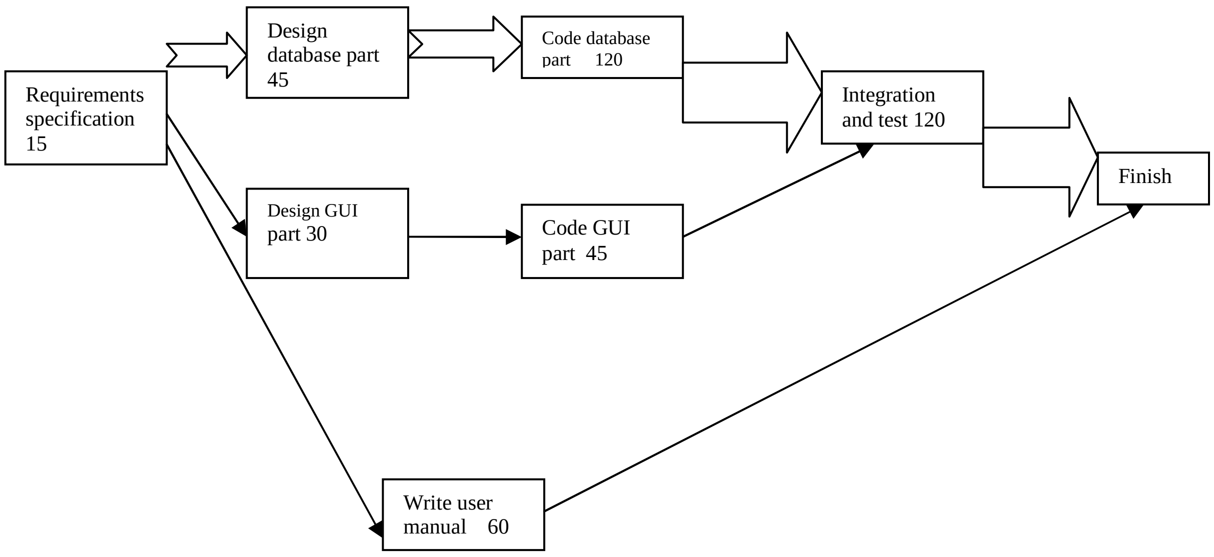


Fig. 2.4 Activity Network representation of the MIS problem

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**
- From the activity network Fig.2.4 representation, the following analysis can be made:
- The minimum time (MT) to complete the project is the maximum of all paths from start to finish.
- The earliest start (ES) time of a task is the maximum of all paths from the start to this task.
- The latest start (LS) time is the difference between MT and the maximum of all paths from this task to the finish.
- The earliest finish time (EF) of a task is the sum of the earliest start time of the task and the duration of the task.

- The latest finish (LF) time of a task can be obtained by subtracting maximum of all paths from this task to finish from MT.
- The slack time (ST) is LS – EF and equivalently can be written as LF – EF. The slack time is the total time for which a task may be delayed before it would affect the finish time of the project. The slack time indicates the flexibility in starting and completion of tasks.
- A critical task is one with a zero slack time.
- A path from the node to the finish node containing only critical tasks is called a critical path.
- The above parameters for different tasks for the MIS problem (Fig.2.4) are shown in the following table.

Task	ES	EF	LS	LF	ST
Specification Part	0	15	0	15	0
Design Database Part	15	60	15	60	0
Design GUI Part	15	45	90	120	75
Code Database Part	60	165	60	165	0
Code GUI Part	45	90	120	165	75
Integrate and Test	165	285	165	285	0
White User Manual	15	75	225	285	210

The critical paths are all the paths whose duration equals MT. The critical path in Fig.2.4 is shown with thick arrow lines.

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.

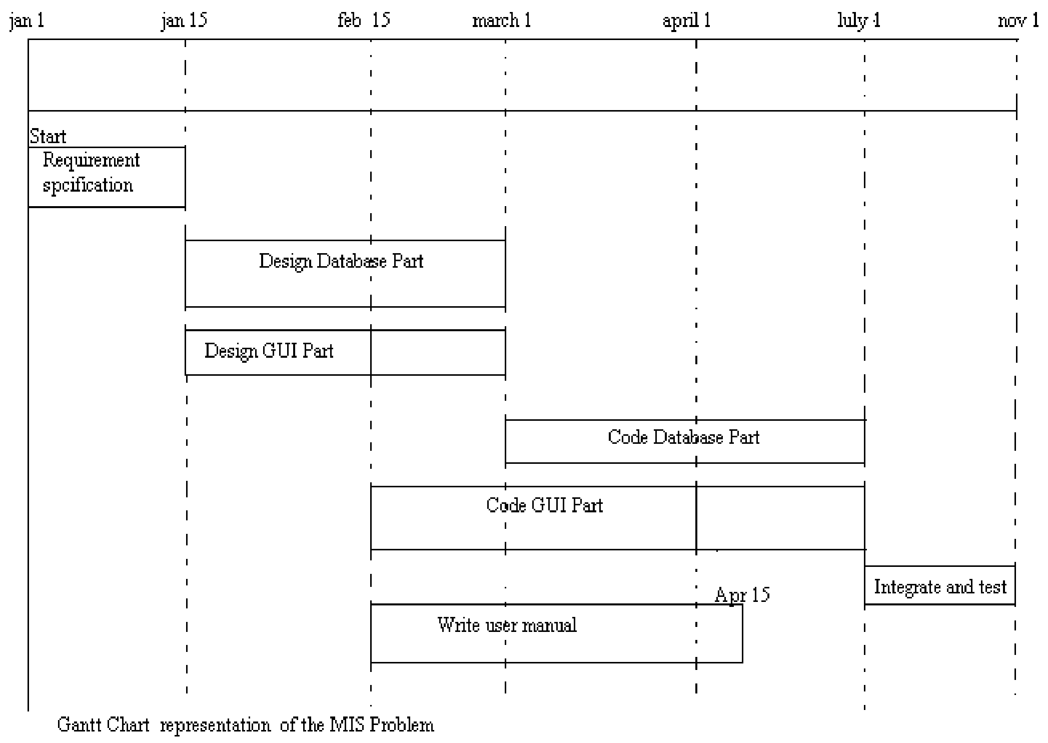


Fig. 2.5 Gantt Chart Representation of the MIS Problem

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.

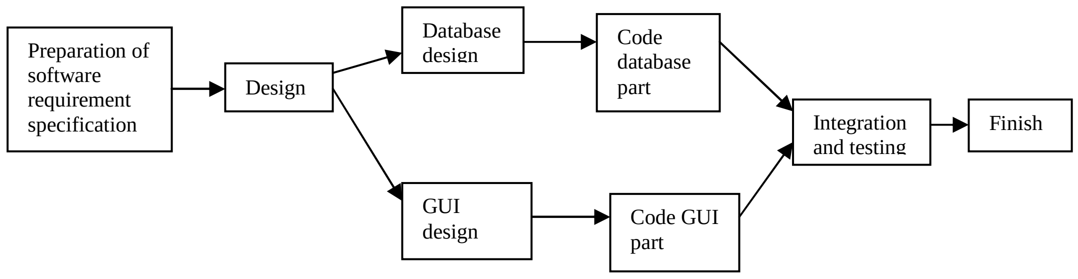


Fig.2.6 PERT Chart representation of the MIS problem

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

2.10 Organisation Structure

There are essentially two broad ways in which a software development organization can be structured: function format and 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 bellow engineers from functional groups for specific phases of the projects and return them to their functional group upon completion of the phase.

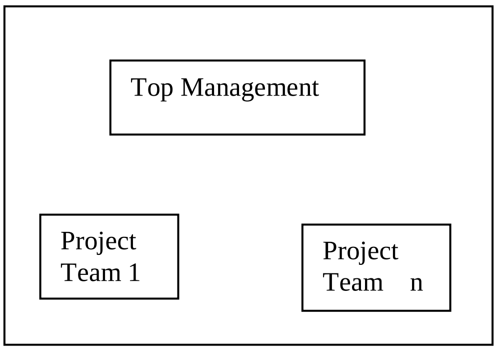


Fig. Project Organization

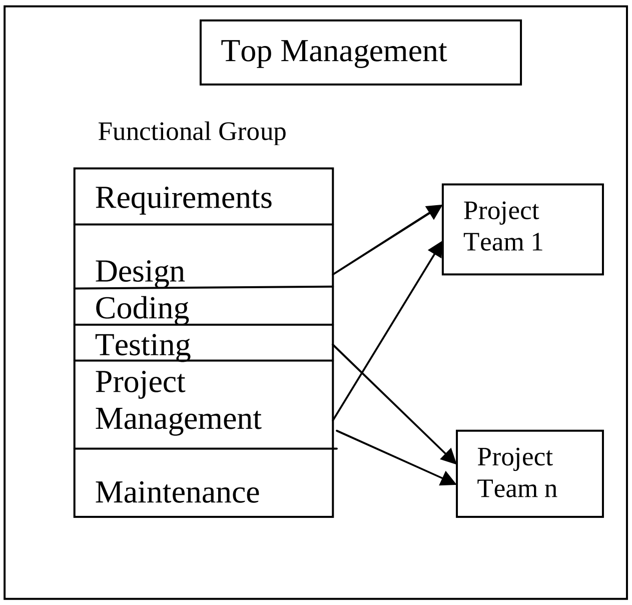


Fig.2.7 Functional organization

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:

- Ease of staffing
- Production of good quality documents
- Job specialization
- 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.

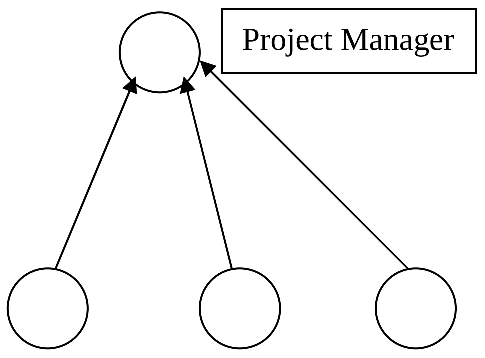
2.11 Team Structure

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.



(Software engineers)
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 intellectual grasp of a single individual.

Democratic Team

The democratic team structure does not enforce any formal team hierarchy. Typically a manager provides the administrative leadership. At different times, different members of the group provide technical leadership.

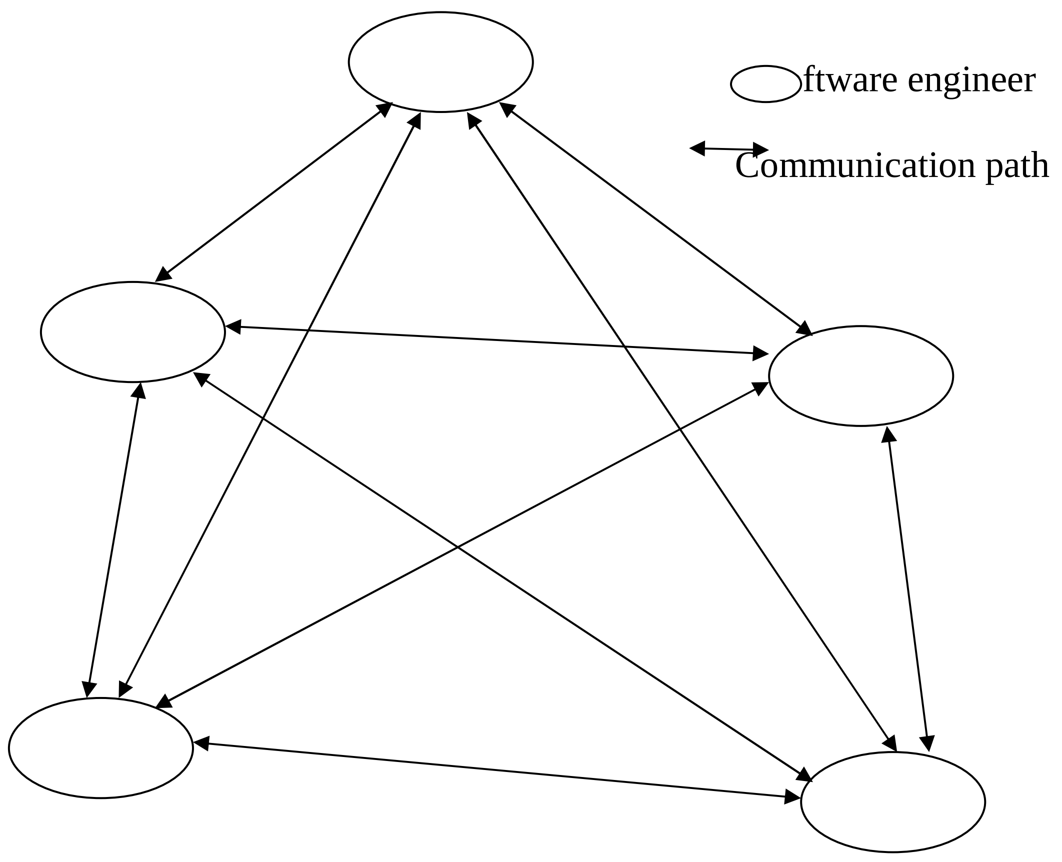


Fig.2.9 Democratic team structure

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.

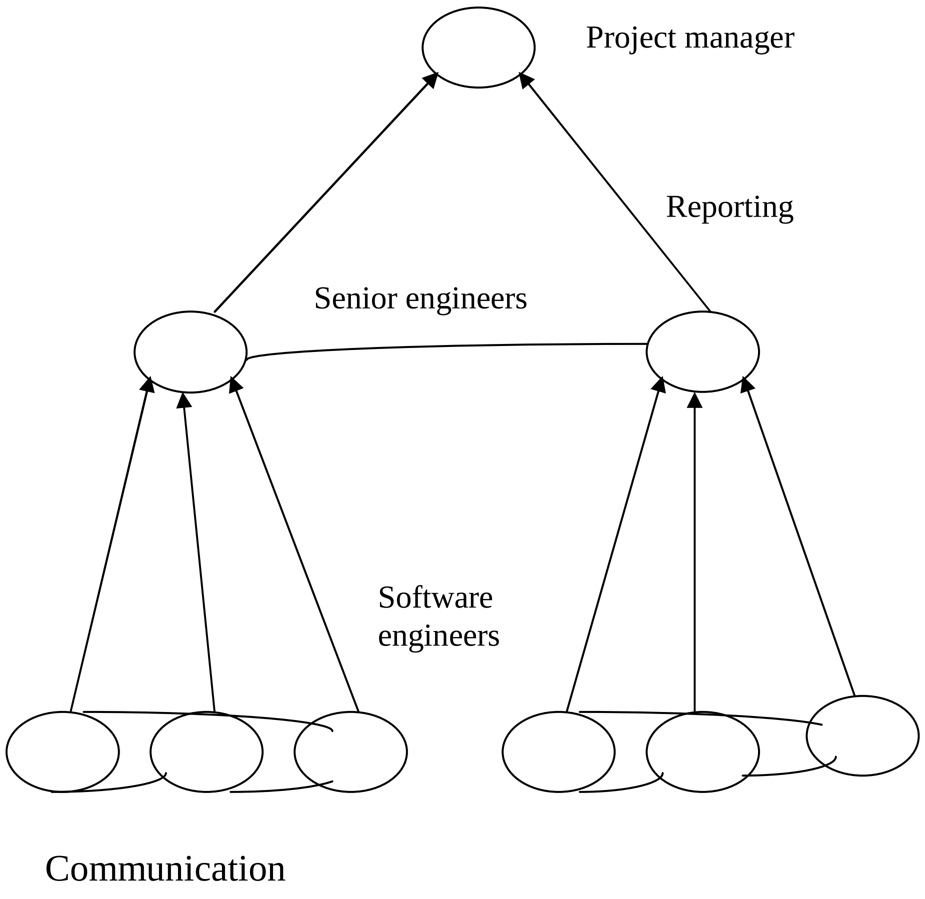


Fig.2.10 Mixed team structure

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.

2.12 Importance of Risk Identification, Risk Assessment and Risk Containment with reference to Risk Management

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

a> The likelihood of a coming true (r)

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

The priority of each risk can be computed as

$$p= r*s$$

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

Risk Containment

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

- Avoid the risk
- Risk reduction
- Transfer the risk

Avoid the Risk

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

Transfer the Risk

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

Risk Reduction

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

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

Chapter-3
Understanding the need of Requirement Analysis

Contents

3.1 Need for Requirement Analysis

Requirement analysis is a Software engineering task that bridges the gap between system level requirements engineering and software design. Requirement analysis provides software designer 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

3.2 Steps in Requirements Elicitation for Software: Initiating the Process, Facilitated Application Specification Techniques, Quality Function Deployment

Before requirements can be analyzed, modeled or specified they must be gathered through an elicitation process.

Initiating the Process

- The most commonly used requirements elicitation technique is to conduct a meeting or interview. Customer meetings are the most commonly used technique.
- Use context free questions to find out customer's goals and benefits, identify stakeholders, gain understanding of problem, determine customer reactions to proposed solutions, and assess meeting effectiveness.

Facilitated Application Specification Techniques

- Meeting held at neutral site, attended by both software engineers and customers.
- Rules established for preparation and participation.
- Agenda suggested to cover important points and to allow for brainstorming.
- Meeting controlled by facilitator (customer, developer, or outsider).
- Goal is to identify problem, propose elements of solution, negotiate different approaches, and specify a preliminary set of solution requirements.

Quality Function Deployment (QFD)

Quality function deployment is a quality management technique that translates the needs of the customer into technical requirements for software. Quality function deployment identifies three types of requirements:

- Normal requirements: The objectives and goals that are stated for a product or system during meetings with the customer.
- Expected requirements: These requirements are implicit to the product or system (customers assumes will be present in a professionally developed product without having to request them explicitly).
- Exciting requirements: These features that go beyond the customer's expectations and prove to be very satisfying when they are present.

Function deployment is used to determine the value of each function that is required for the system. Information deployment identifies both the data objects and events that the system must consume and produce. Task deployment examines the behavior of the system or product within the context of its environment. Value analysis used to determine the relative priority of requirements during function, information, and task deployment.

3.3 Principles of Analysis

All analysis methods are related by a set of operational principles:

- The information domain of the problem must be represented and understood.
- The functions that the software is to perform must be defined.
- Software behavior must be represented

- Models depicting information function and behavior must be partitioned in a hierarchical manner that uncovers detail.
- The analysis process should move from the essential information toward implementation detail.

3.4 Software Prototyping

The prototyping paradigm can be either close-ended or open-ended. The close-ended approach is called throwaway prototyping and an open-ended approach called evolutionary prototyping.

3.5 Prototyping Approach

Throwaway prototyping: Prototype only used as a demonstration of product requirements.

Evolutionary prototyping uses the prototype as the first part of an analysis activity that will be continued into design and construction.

The customer must interact with the prototype, it is essential that:

a) Customer resources must be committed to evaluation and refinement of the prototype.
b) Customer must be capable of making requirements decisions in a timely manner.

3.6 Prototyping Tools and Methods

Three generic classes of methods and tools are:

- Fourth generation techniques: Fourth generation techniques (4GT) tools allow software engineer to generate executable code quickly.
- Reusable software components: Assembling prototype from a set of existing software components.
- Formal specification and prototyping environments can interactively create executable programs from software specification models.

3.7 Software Requirement Specification Principle

Specification principles are:

- Separate functionality from implementation.
- Develop a behavioral model that describes functional responses to all system stimuli.
- Define the environment in which the system operates and indicate how the collection of agents will interact with it.
- Create a cognitive model rather than an implementation model
- Recognize that the specification must be extensible and tolerant of incompleteness.
- Establish the content and structure of a specification so that it can be changed easily.

3.8 SRS Document

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

- Requirements gathering and analysis.
- Requirements specification

System analysts collect data pertaining to the product to be developed and analyze these data to conceptualize what exactly needs to be done. The analyst starts the requirements gathering and analysis activity by the collecting all information from the customer which could be used to develop the requirements of the system. The analyst then analyzes the collect information to obtain a clear and thorough understanding of the product to be developed.

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

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

There are three main types of problems in the requirement that analyst needs to identify and resolve:

- Anomaly
- Inconsistency
- Incompleteness

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

Example: In a process control application, a requirement expressed by one user is that when the temperature becomes high, the heater should be switched off. (Words such as high, low, good, bad etc, are ambiguous without proper quantification). If the threshold above which the temperature can be considered to be high is not specified, then it can be interpreted differently by different people.

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

Example: For the case study of the office automation, one of the clerk described that a student securing fail grades in three or more subjects should have to repeat the entire semester. Another clerk mentioned that there is no provision for any student repeat a semester.

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

Software Requirement Specification

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

- The software developers refer to the SRS document to make sure that they develop exactly what is required by the customer.
- Test Engineers: Their goal is to ensure that the requirements are understandable from a functionality point of view, so that they can test the software and validate its working.
- User Documentation Writers: Their goal in reading the SRS document is to ensure that they understand the document well enough to be able to write the users' manuals.
- Project Managers

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

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

Contents of the SRS Document

An SRS document should clearly document:

- Functional Requirements
- Nonfunctional Requirements
- Goals of implementation

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.

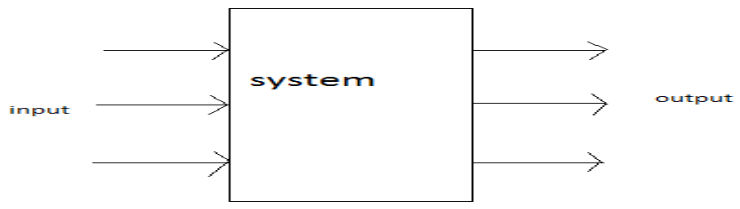


Fig. 3.1 Contents of SRS Document

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

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

3.9 Characteristics and Organization of SRS Document

Characteristics of SRS document

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

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

Block-box View: It should specify what the system should do. The SRS document should specify the external 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 the SRS Document

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

1. Introduction

(a) Background

(b)Overall Description

(c)Environmental Characteristics

   (i)Hardware

   (ii)Peripherals

   (iii)People

1. Goals of implementation

Functional requirements

Nonfunctional Requirements

Behavioural Description

(a) System States

(b)Events and Actions

The `introduction' section describes the context in which the system is being developed, an overall description of the system and the environmental characteristics. The environmental characteristics subsection describes the properties of the environment with which the system will interact.

Chapter-4

Understanding the Principles and Methods of S\W Design

Contents

4.1 Importance of Software Design

Software design aims to plan and create a blueprint for the implementation of the software. The main aim and focus of the software design process is to cover the gap between understanding the specification and implementing them in the software. Software design transforms the SRS document into implementable form using a programming language. The design representations are used to describe how the system is to be structured and developed to meet the specification in the best manner.

The following items are designed and documented during the design phase.

- Different modules in the solution should be cleanly identified. Each module should be named according to the task it performs.
- The control a relationship exists among various modules should be

identified in the design document. The relationship is also known as the call relationship.

- Interface among different modules. The interface among different modules identifies the exact data items exchanged among the modules.
- Data structures of the individual modules.
- Algorithms required to implement the individual modules.

4.2 Design Principles and Concepts

Design Principles

Software design is both a process and a model. The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built. Basic design principles are:

- The design process should not suffer from “tunnel vision”.
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should “minimize the intellectual distance” between the software and the problem in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently.
- Design is not coding.
- The design should be assessed for quality.
- The design should reviewed to minimize conceptual errors.

Design Concepts

Abstraction: Each step in the software engineering process is a refinement in the level of abstraction of the software solution.

- Data abstractions: a named collection of data

- Procedural abstractions: A named sequence of instructions in a specific function
- Control abstractions: A program control mechanism without specifying internal details.

The design process takes the SRS documents as the input and is dedicated to plan for implementation of the software. The design activities are classified into two parts.

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

Preliminary Design / High-Level Design

Through high-level design, a problem is decomposed into a set of modules, the control relationships among various modules identified and also the interfaces among various modules are identified. The outcome of high-level design is called the program structure or the software architecture many different types of notations have been used to represent a high-level design. A popular way is to use a tree-like diagram called the structured chart to represent the control hierarchy in high-level design. Another popular design representation technique called UML that is being used to document object-oriented design. Once the high-level design is complete, detailed design is undertaken.

Detailed Design

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

What is a Good Software Design

There is no unique way to design a system. Using the same design methodology, different engineers can arrive at very different design solutions. Even the same engineer can work out many different solutions to the same problem.

The definition of “a good software design” can vary depending on the application for which it is being designed. For example, the memory size used up by a program may be an important issue to characterize a good solution for embedded software development-since embedded applications are often required to be implement using memory of limited size due to space, cost or power consumption constraints. For embedded applications, factors such as design comprehensibility may take a back seat while judging the goodness of design. For embedded applications, one may sacrifice design comprehensibility to achieve code compactness. Therefore, the criteria used to judge how good a given design solution is can vary widely depending on the application. 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.

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

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

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

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

Modularity

A modular design achieves effective decomposition of a problem. It is a basic characteristic of any good design solution. Decomposition of a problem into modules facilitates the design by taking advantage of the divide and conquers principle. If different modules are independent of each other, then each module can be understood separately. This reduces the complexity of the design solution.

Clean Decomposition

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

Layered Design

In a layered design, the modules are arranged in a hierarchy of layers. A module can only invoke functions of the modules in the layer immediately below it. A layer design can make the design solution easily understandable. A layer design can be considered to be implementing control abstraction, since a module at a lower layer is unaware of the higher layer modules.

4.3 Cohesion and Coupling

A good software design implies clean decomposition of the problem into modules and thereafter the neat arrangement of these modules. The primary characteristics of a neat module decomposition are high cohesion and low coupling. Cohesion is a measure of the functional strength of a module where as the coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. A modules having high cohesion and low coupling is said to be functionally independent of other modules. A cohesive module performs a single task or function.

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. High cohesion facilitates execution of a task by maximum intra-modular communication and least inter-module dependencies. It promotes independencies between different modules.

Error isolation

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

Scope for Reuse

Reuse of a module becomes possible, because each module does some well-defined and precise functions and the interface of the module with other module is simple and minimal. Therefore a cohesive module can be easily taken out and be reused in a different program.

Understandability

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

4.4 Classification of Cohesiveness

There are seven types or levels of cohesion.

Coincidental	Logical	temporal	Procedural	communication	sequential	functional

Low $\longrightarrow$ High

Fig. 4.1 Classification of Cohesion

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

Coincidental Cohesion

A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case the module contains a random collection of functions.

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

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.

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.

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.

Communication Cohesion

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

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.

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.

4.5 Classification of 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.

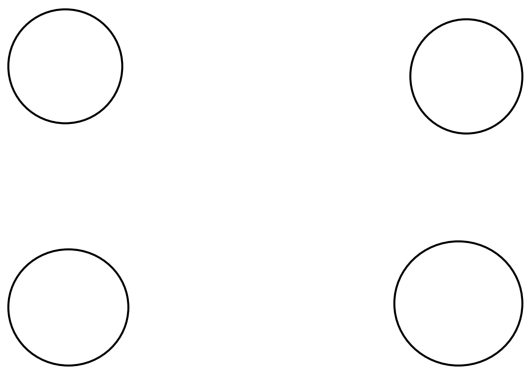


Fig. 4.2 Uncoupled: No Dependencies

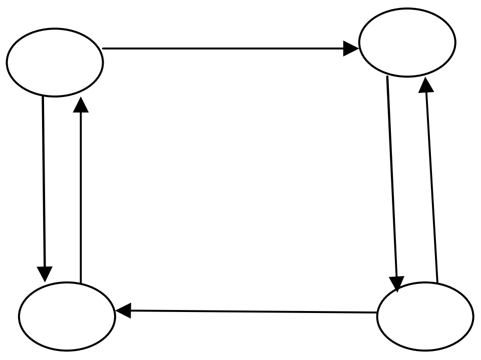


Fig. 4.3 Loosely coupled: some dependencies

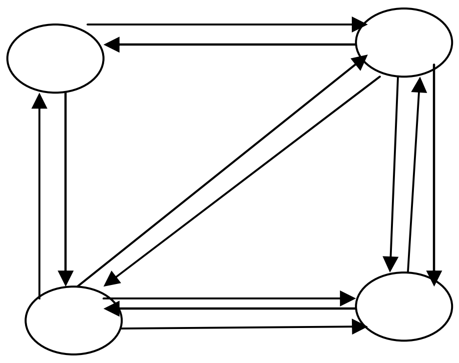


Fig. 4.4 Highly coupled: many dependencies

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 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.

Data	Stamp	Control	Common	Content

Fig. 4.5 Classification of Coupling

Stamp Coupling

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

Control Coupling

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

Common Coupling

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

Content coupling

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

4.6 S/W Design Approaches

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

Function oriented design

Features of the function-oriented design approach are:

Top-down decomposition

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

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

- assign-membership-number
- create-member-record
- print-bill

  Each of these sub functions may be split into more detailed sub functions and so on.

Object Oriented Design

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

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.

4.7 Structured Analysis Methodology

The aim of structured analysis activity is to transform a textual problem description into a graphic model. Structured analysis is used to carry out the top-down decomposition of the set of high-level functions depicted in the problem description and to represent them graphically. During structured design, all functions identified during structured analysis are mapped to a module structure. Structure analysis technique is based on the following principles:

- ✓ Top-down decomposition approach
- ✓ Divide and conquer principle. Each function is decomposed independently

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

4.8 Use of Data Flow Diagram

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

4.9 Lists the Symbols used in DFD

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

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

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

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

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

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

4.10 Construction of DFD

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

A DFD start with the most abstract definition of the system (lowest level) and at each higher level DFD, 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.

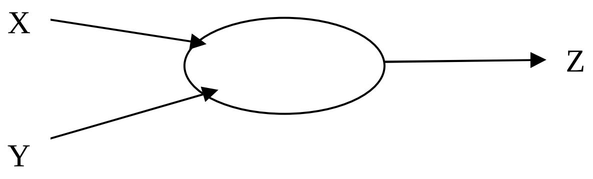


Fig. 4.6 Context Diagram

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 on until a level is reached at

Example: Student admission and examination system

This statement has three modules, namely

- Registration module
- Examination module
- Result generation module

Registration module:

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

Examination module:

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

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

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

Result generation Module:

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

Data Flow Diagram

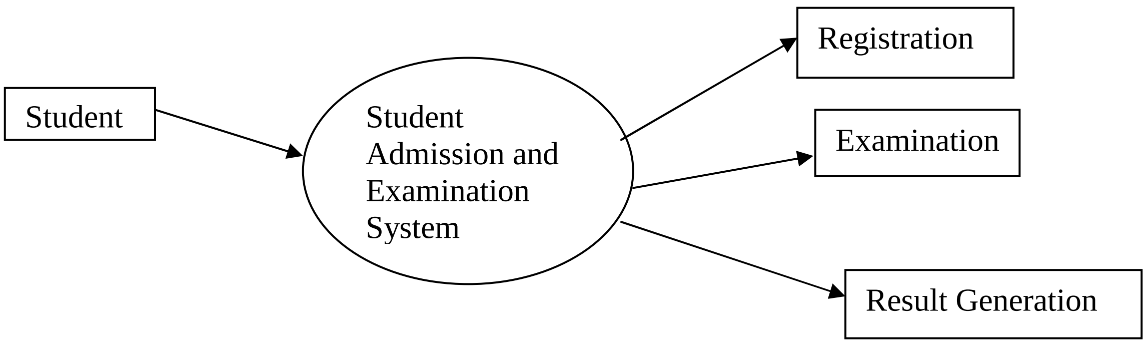


Fig. 4.7 Level 0 DFD or Context Diagram

Level 1 DFD

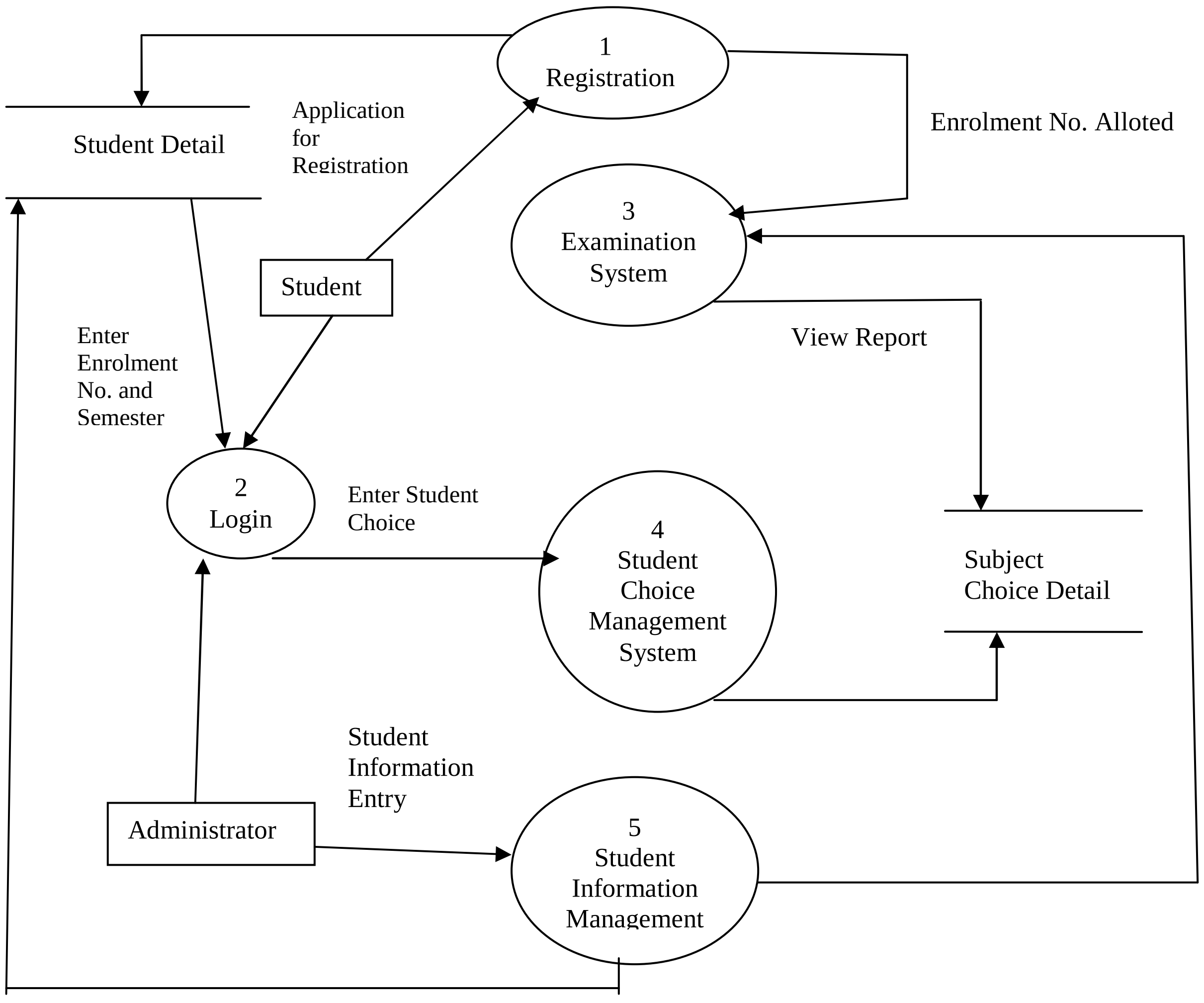

1 Registration
Student Detail
Application for Registration
Enrolment No. Alloted
3 Examination System
Student
View Report
Enter Enrolment No. and Semester
2 Login
Enter Student Choice
4 Student Choice Management System
Subject Choice Detail
Student Information Entry
Administrator
5 Student Information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

Fig 4.8 Level 1 DFD of Student Admission and Examination System

Level 2 DFD

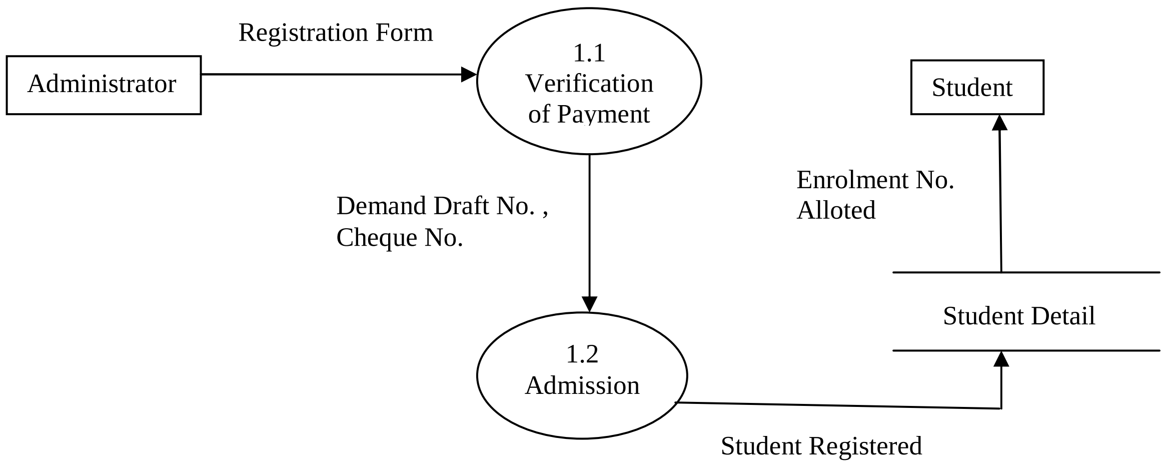


Fig.4.9 Level 2 DFD of Registration

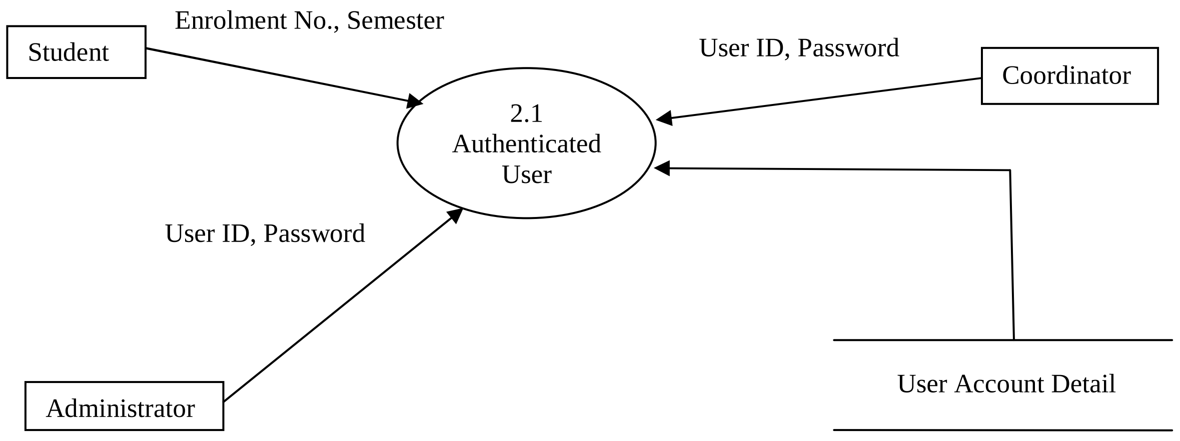


Fig. 4.10 Level 2 DFD of Authenticated

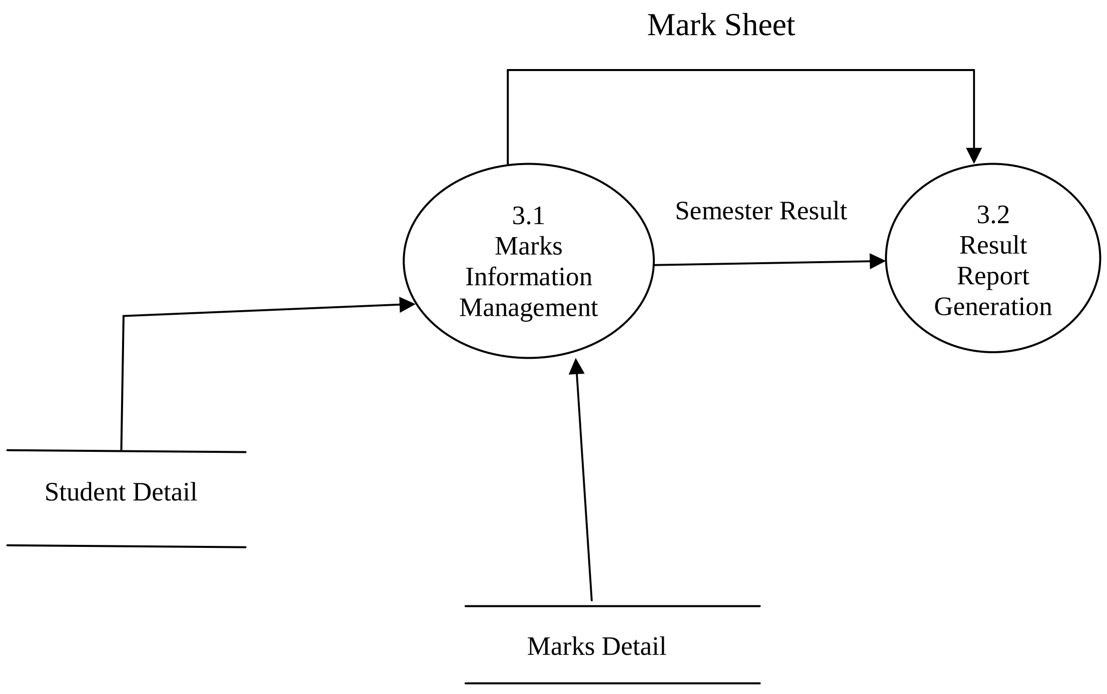


Fig 4.11Level 2 DFD of Examination

4.11 Limitations of DFD

- A data flow diagram does not show flow of control. It does not show details linking inputs and outputs within a transformation. It only shows all possible inputs and outputs for each transformation in the system.
- The method of carrying out decomposition to arrive at the successive 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.

4.12 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.

4.13 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

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. The first level of structure chart is produced by representing each input and output unit as boxes and each central transforms a single box.

In the third step of transform analysis, the structure chart is refined by adding subfunctions required by each of the high-level functional components. Many levels of functional components may be added. This process of breaking functional components into subcomponents is called factoring. Factoring includes adding read and write modules, error-handling modules, initialization and termination process etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

Transaction Analysis

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

Chapter-5
Understanding the Principles of User Interface Design

5.1 Rules for UID (User Interface Design)

User interface design creates an effective communication medium between a human and a computer. User interface design begins with the identification of user, task, and environment requirements. Once user tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions.

Three Golden rules of user interface design are:

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

Place the user in control

Number of design principles that allow the user to maintain the control are:

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
- Provide for flexible interaction, different users have different interaction preferences.
- Allow user interaction to be interruptible.

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

Reduce the User's Memory Load

Principles that enable an interface to reduce the user's memory load are:

- Reduce demand on short-term memory.
- Establish meaningful defaults.
- Define shortcuts that are intuitive.
- Disclose information in a progressive fashion.

Make the Interface Consistent

The interface should present and acquire information in a consistent fashion. The set of design principles that help make the interface consistent are:

- Allow the user to put the current task into a meaningful context.
- Maintain consistency across a family of applications.

5.2Interface Design Models

The process for designing a user interface begins with the creation of different models of system function. Four different user interface design models are:

- ❖ User model
- ❖ Design model
- ❖ Mental model
- ❖ Implementation model

A software engineer establishes a user model, the software engineer creates a design model, the end-user develops a mental image that is often called the user's model or the system perception, and the implementation of the system create a system image.

A design model of the entire system incorporates data, architectural, interface, and procedural representations of the software.
The user model establishes the profile of end-users of the system. The system perception is the image of the system that end-users carry in their heads.

5.3 The User Interface Design Process

The design process for user interfaces is iterative and can be represented using a spiral model. The user interface design process encompasses four distinct activities

- User, task, and environment analysis and modelling
- Interface design
- Interface construction
- Interface validation

The initial analysis activity focuses on the profile of the users who will interact with the system. Skill level and business understanding are recorded and different user categories are defined. The software engineer attempts to understand the system perception for each class of users.
Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described and elaborated.
The goal of interface design is to define a set of interface objects and actions that enable a user to perform all defined tasks that meets every usability goal defined for the system.

5.4 Interface Design Activities, Defining Interface Objects and Actions and the Design Issues

Interface Design Activities

Once task analysis has been completed, all tasks required by the end-user have been identified and the interface design activity commences. Interface design steps can be accomplished using the following approach:

- Establish the goals and intentions for each task.
- Map each goal and intention to a sequence of specific actions.
- Specify the action sequence of tasks and subtasks.
- Indicate the state of the system.
- Define control mechanisms, that is the objects and actions available to the user to alter the system state.
- Show how control mechanisms affect the state of the system.
- Indicate how the user interprets the state of the system from information provided through the interface.

Defining Interface Objects and Actions

Once the objects and actions have been defined and elaborated. Interface objects are categorized into types: source, target, and application:

- A source object (e.g. a report icon) is dragged and dropped onto a target object (e.g. a printer icon) such as to create a hard copy of the report.
- An application object represents application-specific data that are not directly manipulated as part of screen interaction such as a list.

After identifying objects and their actions, an interface designer performs screen layout which involves:

  - Graphical design and placement of icons

- Definition of descriptive screen text
- Specification and titling for windows
- Definition of major and minor menu items

Design Issues

Four common design issues are:

- System response time
- User help facilities
- Error information handling and
- Command labelling

System response time is the primary complaint for many interactive applications. System response time is measured from the point at which the user performs some control action until the software responds with desired output or action. Two important characteristics of system response time are length and variability.

Two different types of help facilities are integrated and add-on. An integrated help facility is designed into the software from the beginning. An add-on help facility is added to the software after the system has been built. User help facilities must be addressed: when it is available, how it is accessed, how it is represented to the user, how it is structured, what happens when help is exited.

An effective error message can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur. Every error message or warning produced by an interactive system should have the following characteristics:

- The message should describe the problem in simple language that a user can easily understand.
- The message should provide constructive advice for recovering from the error.

- The message should indicate any negative consequences of the error.
- The message should be accompanied by an audible or visual cue such as a beep, momentary flashing, or a special error colour.

5.5 Compare the Various Types of Interface

User interfaces broadly classified into three categories:

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

Command Language-Based Interfaces

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

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

Issues in Designing a Command Language Interface

- The designer has to decide what mnemonics to use for the different commands. The designer should try to develop meaningful mnemonics and yet be concise to minimize the amount of typing required.
- The designer has to decide whether the user will be allowed to redefine the command names to suit their own preferences.
- The designer has to decide whether it should be possible to compose primitive commands to form more complex commands. A sophisticated command composition facility would require the syntax and semantics of the various command composition options to be clearly and unambiguously specified. The ability to combine commands can be usefully exploited by experienced users, but is quite unnecessary for inexperienced users.

Menu-based interfaces

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

Experienced users find a menu-based user interface to be slower than a command language-based interface because they can type fast and get speed advantage by composing different primitive commands to express complex commands. Composing commands in a menu-based interface is not possible.

A major challenge in the design of a menu-based interface is that of structuring the large number of menu choices into manageable forms.

The techniques available to structure of menu items are:

Scrolling Menu

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

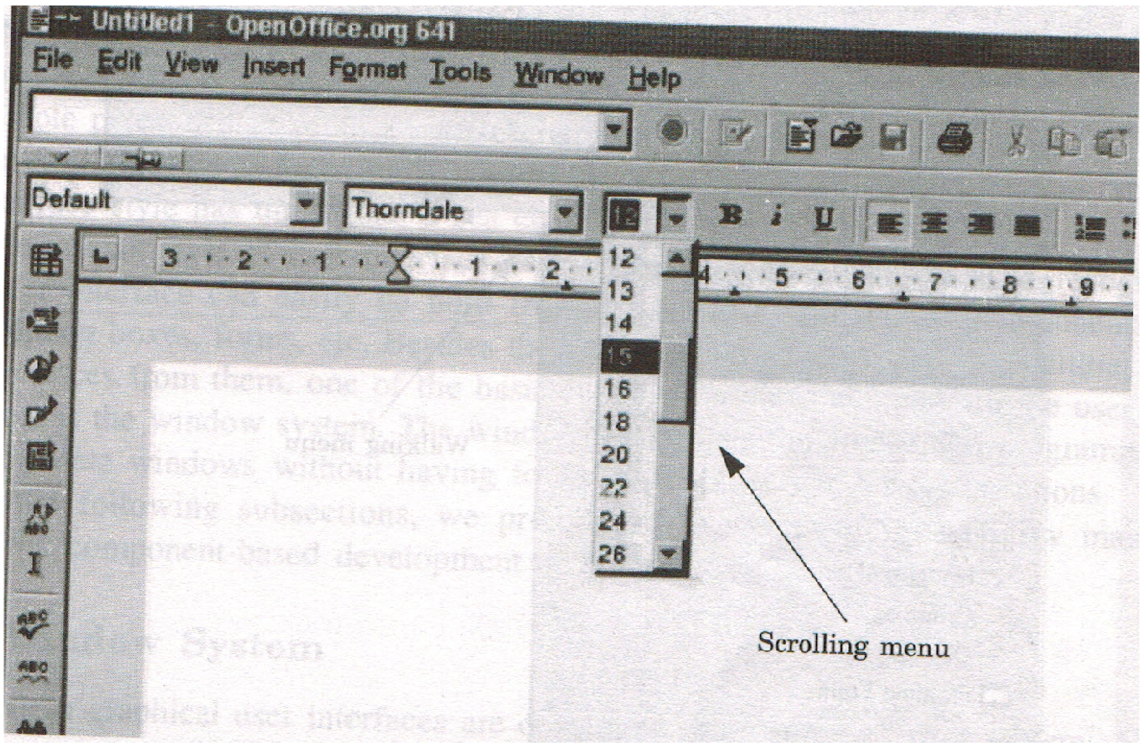

Untitled1 - OpenOffice.org 641
File Edit View Insert Format Tools Window Help
Default
Thorndale
12
13
14
15
16
18
20
22
24
26
Scrolling menu

Fig.5.1 Font size selecting using scrolling menu

Walking Menu

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

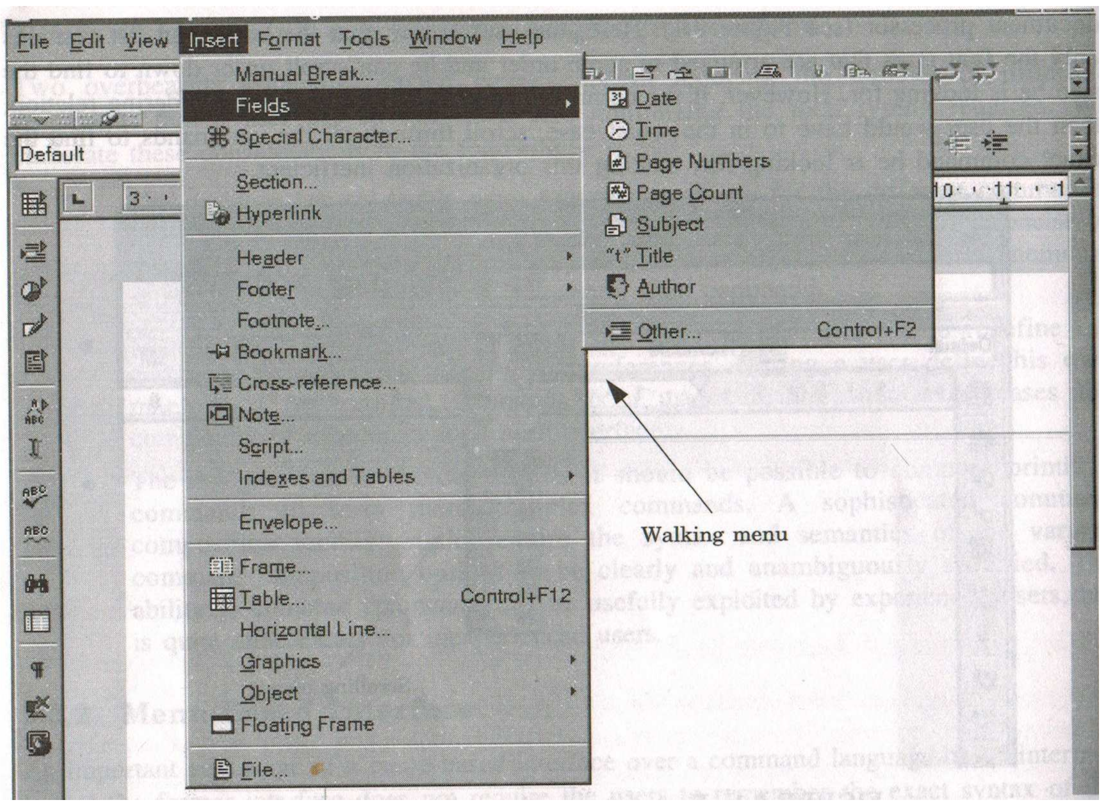


Fig.5.2 Examples of walking menu

Hierarchical Menu:

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

Direct Manipulation Interfaces

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

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

5.6 Main aspects of Graphical UI, Text based Interface

Aspects of GUI

- In a GUI, multiple windows with different information can simultaneously be displayed on the user screen.
- Iconic information representation and symbolic information manipulation is possible in a GUI. Symbolic information manipulation, such as dragging an icon representing a file to a trash can for deleting, is intuitively very appealing and the user can instantly remember it.
- A GUI usually supports command selection using an attractive and user-friendly menu selection system.
- In a GUI, a pointing device such as a mouse or a light pen can be used for issuing commands. The use of a pointing device increases the efficacy of the command issue procedure.
- A GUI flip side, a GUI requires special terminals with graphics capabilities for running and also requires special input devices such as a mouse.

Text Based Interface

Text based interface only use text, symbols and colours available on a given text environment. Text- based user interface can be implemented on a cheap alphanumeric display terminal.

Chapter -6

Understanding the Principles of Software Coding

6.1 Coding Standards and Guidelines

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

Representative coding standards are:

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

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

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

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

6.2 Code Walk-Through

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

In this technique, after a module has been coded, it is successfully compiled and all syntax errors are eliminated. Some members of the development team are given the code a few days before the walk-through meeting to read and understand the code. Each member selects some test cases and simulates execution of the code through different statements and functions of the code.

Even though a code walkthrough is an informal analysis technique, several guidelines have evolved for making this technique more effective and useful. Some guidelines are:

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

6.3 Code Inspection and Software Documentation`

Code Inspection

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

- ✓ Use of uninitialized variables
- ✓ Jumps into loops
- ✓ Non-terminating loops
- ✓ Array indicates out of bounds
- ✓ Improper storage allocation and deallocation
- ✓ Use of incorrect logical operators
- ✓ Improper modification of loop variables
- ✓ Comparison of equality of floating point values.

Software Documentation`

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

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

- ➢ Good documents help the users in effectively exploiting the system.
- ➢ Good documents help in effectively overcoming the manpower turnover problem. Even when an engineer leaves the organization, the newcomer can build up the required knowledge quickly.
- ➢ Good documents help the manner in effectively tracking the progress of the project.

Different types of software documents can be broadly classified into:

- 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.

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

A software product is normally tested in the three levels:

- Unit testing
- Integration testing
- System testing

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

An integration test is done to demonstrate that different pieces of the system work together. Integration tests cover whole applications, and they require much more effort to put together. They usually require resources like database instances and hardware to be allocated for them. The integration tests do a more convincing job of demonstrating the system works (especially to non-programmers) than a set of unit tests .

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

6.5 Unit Testing

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

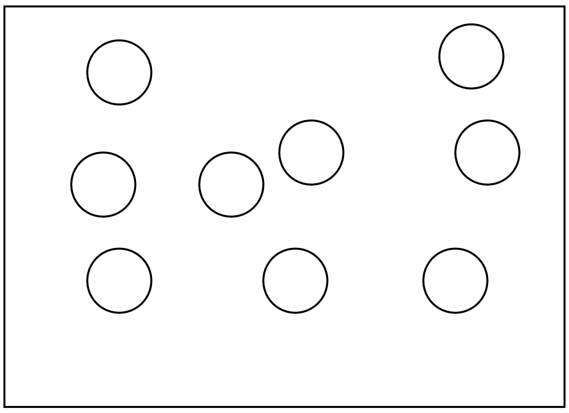


Fig. 6.1 Unit testing

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:

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

Driver and Stub Modules

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

- The procedures belonging to other modules that the module under test calls.
- Nonlocalb data structures that the module accesses.
- A procedure to call the function of the module under test with appropriate parameters.

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

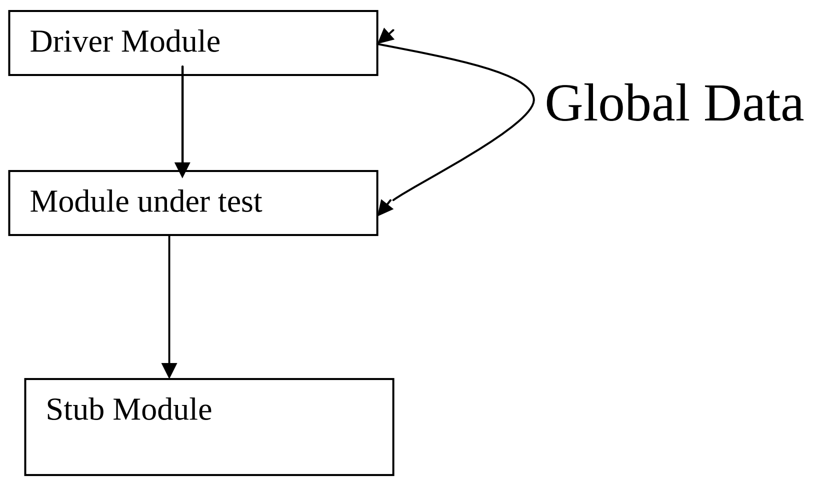


Fig. 6.2 Unit testing with the help of driver and stub module

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

6.6 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

6.7 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. Guidelines for designing the equivalence classes are:

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

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

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).

So accordingly the following test cases are designed

Test case1=(-5,3000,7001), Test case2=(-20,100,5050),

Test case3=(-6,4000,9000)

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}

Summary of the Black-box test suite Design

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

6.8 Methodologies for 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

6.9 Different white box methodologies: statement coverage branch coverage, condition coverage, path coverage, data flow based testing and mutation testing.

Statement Coverage

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

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)}.

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)}.

Condition Coverage

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

Path Coverage

The path coverage-based testing strategy requires designing test cases such that all linearly independent paths 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.

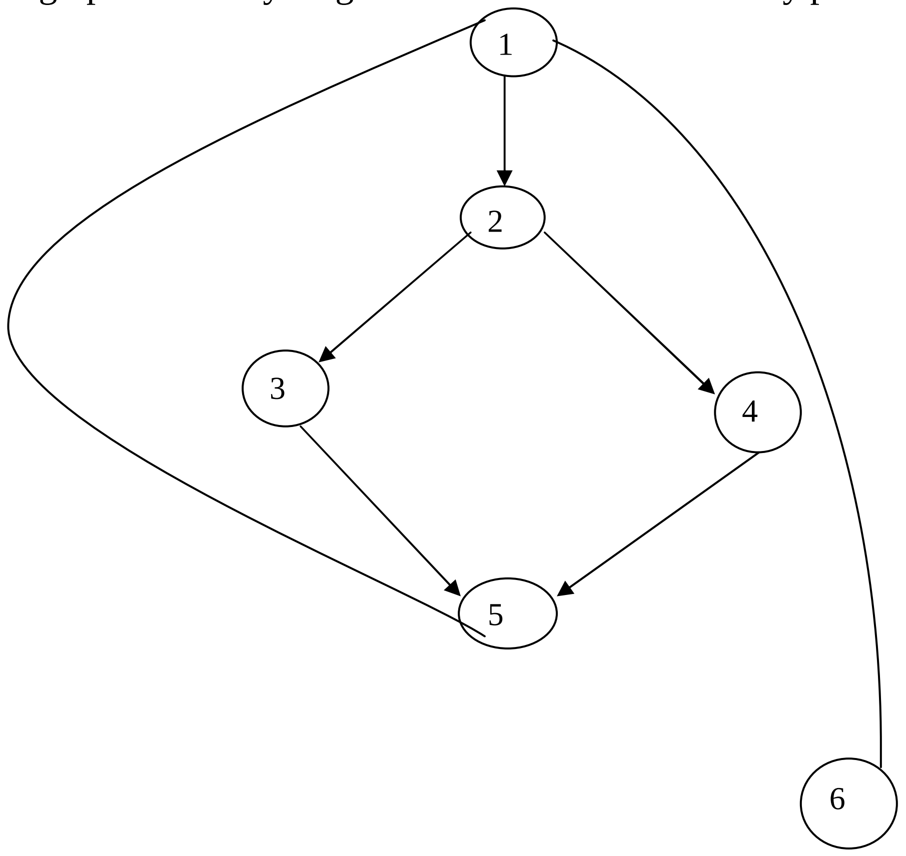


Fig. 6.3 Control Flow Graph

```
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:

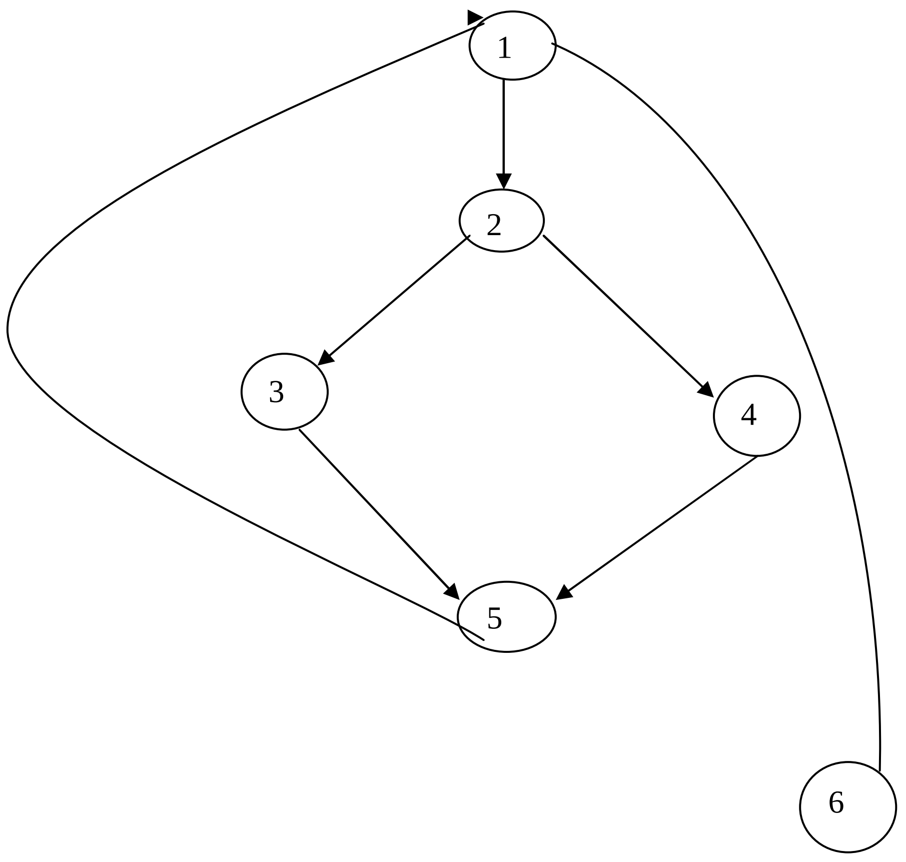


Fig. 6.4 Control Flow Graph

Number of Edges = E = 7
Number of Nodes = N = 6
The value of cyclomatic complexity is

$$\begin{aligned} V(G) &= E - N + 2 \\ &= 7 - 6 + 2 \\ &= 3 \end{aligned}$$

Data Flow – Based Testing

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

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

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

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

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

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

Mutation Testing

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

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

6.10 Debugging Approaches

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.

6.11 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.

6.12 Need for Integration Testing

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

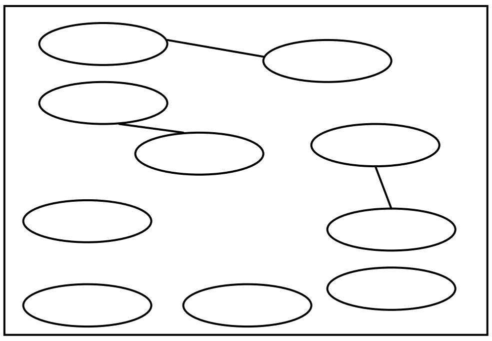


Fig.6.5 Integration Testing

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

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

Big – Bang Approach

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

Top – Down Approach

Top – down integration 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.

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.

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.

6.13 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 nonfunctional requirements of the system.

Performance Testing

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 nonfunctional requirements of the system document in the SRS document. All performance tests can be considered as black – box tests.

6.15 Need for Stress Testing and Error Seeding

Stress Testing

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

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

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

Volume Testing

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

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

Configuration Testing

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

Compatibility Testing

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

Regression Testing

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

Recovery Testing

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

Maintenance Testing

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

Documentation Testing

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

Usability Testing

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

Error Seeding

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 is 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.

$n / N = s / S$

$\Rightarrow N = S * n / s$

$\Rightarrow (N-n) = n(S-s) / S$

6.16 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
Understanding the Importance of S/W Reliability

7.1 Importance of Software Reliability

Reliability of a software product can be defined as the probability of the product working correctly over a given period of time. A software product having a large number of defects is unreliable. 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.

Software Reliability and Hardware Reliability

Reliability behavior for hardware and software is very different. Hardware failures are due to component wear and tear. If hardware failure occurs one has to either replace or repair the failed part. A software product would continue to fail until the error is tracked down and either the design or the code is changed. For this reason, when level that existed before the failure accrued, whereas when a software failure is repaired, the reliability nay either increase or decrease.

There are three phases in the life of any hardware component i.e. burn in, useful life and wear out.

In burn in phase, failure rate is quite high initially as it starts decreasing as the faulty components are identified and removed. The system then enters its useful life.

During useful life period, failure rate is approximately constant. Failure rate increases in wear- out phase due to warning out components. The best period is useful life period. The shape of this curve a “both- tub” and it is also known as both tub curve.

For software the failure rate is highest during integration and testing phases.

During the testing phase more and more errors are identified and moved resulting in a reduced failure rate. This errors removal continues at a slower speed during the useful life of the product. As the software becomes absolute, no more error correction occurs and the failure rate remains unchanged.

7.2 Distinguish between the Different Reliability Metrics

The reliability requirements for different categories of software products may be different for this reason, it is necessary that the level of reliability required for a software product should be specialized in the SRS document. Some reliability metrics which can be used to quantity the reliability of software products are:

- **Rate of Occurrence of Failure (ROCOF)**

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

- **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.

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.

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.

Software Quality

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

Portability

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

Reusability

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

Correctness

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

Maintainability

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

7.3 Reliability Growth Modeling

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.

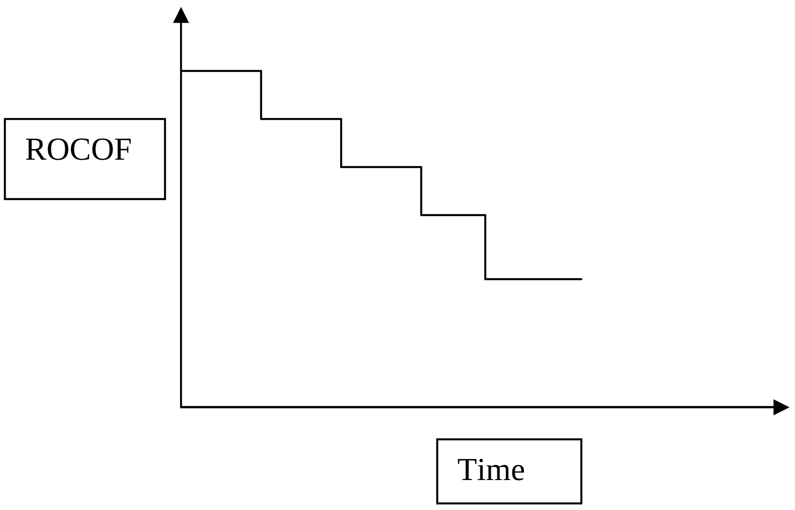


Fig.7.1 Step function model of reliability growth

Littlewood and Verall's Model

This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors. It also models the fact that as errors are repaired, the average improvement in reliability per repair decreases. It treats an error's contribution to reliability improvement to be an independent random variable having gamma distribution. This distribution models the fact that error corrections with large contributions to reliability growth are removed first. This represents diminishing return as test continues.

7.4 Characteristics of Quality Software

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.

7.5 Evolution of 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.

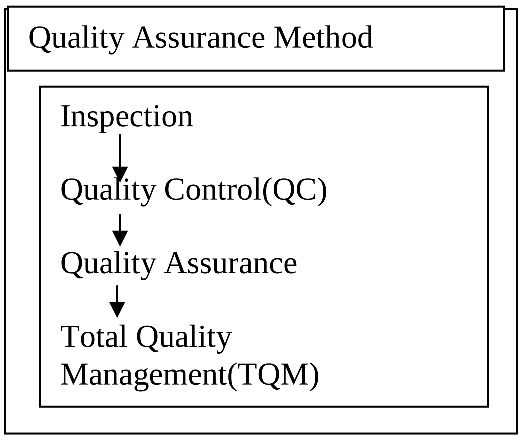
However, many organization have a separate quality department to perform several quality system activities. The quality system of an organization should have the support of the top management

- **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 :



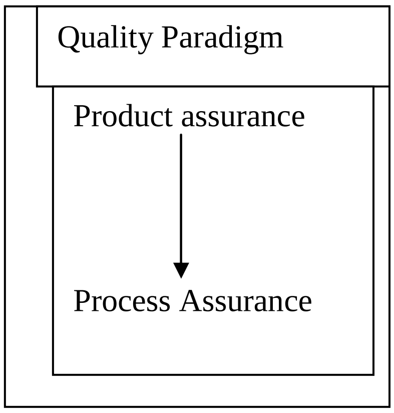


Fig. 7.2Evolution of quality system and the corresponding shift in the quality paradigm.

- 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.

7.6 Importance, Requirement and Procedure to Gain ISO 9000 Certification for Software Industry

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

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

ISO 9000 is a series of three standards: ISO 9001, ISO 9002, and ISO 9003.

ISO 9001: This standard applies to the organisations engaged in design, development, production, and servicing of goods. This standard is applicable to most software development organisations.

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

ISO 9003: This standard applies to organisations involved only in installation and testing of the products.

Requirement of ISO 9000 Certification

- ❖ Confidence of customers in an organisation increases when the organisation qualifies for ISO 9001 certification.
- ❖ ISO 9000 requires a well-documented software production process.
- ❖ ISO 9000 makes the development process focused, efficient, and cost-effective.
- ❖ ISO 9000 certification points out the weak points of an organization and recommends remedial action.
- ❖ ISO 9000 sets the basic framework for the development of an optimal process.

Procedure to gain ISO 9000 Certification

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

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

7.7 SEI Capability Maturity Model (SEI CMM)

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

Level 1: Initial. A software development organization at this level is characterized by ad hoc activity. Very few or no processes are defined and followed. Since software production processes are not defined, different engineers follow their own process and as a result the development efforts become chaotic. It is called chaotic level.

Level 2: Repeatable. At this level, the basic project management practices such as tracking cost and schedule are established. Size and cost estimation techniques like function point analysis, COCOMO etc. are used.

Level 3: Defined. At this level, the processes for both management and development activities are defined and documented. There is a common organization-wide understanding of activities, roles and responsibilities. The processes though defined, the process and the product qualities are not measured. ISO 9000 aims at achieving this level.

Level 4: Managed: At this level, the focus is on software metrics. Two types of metrics are collected. Product metrics measure the characteristics of the product being developed, such as its size, reliability, time complexity, understandability etc. Process metric reflect the effectiveness of the process being used, such as the average defect correction time, productivity, the average number of defects found per hour of inspection, the average number of failures detected during testing per LOC, and so forth

Level:5 Optimizing: At this stage, the process and the product metrics are collected. Process and Product measurement data are analyzed for continuous process improvement.

7.8 Compare between ISO 9000 Certification and SEI/CMM

- ♦ ISO 9000 is awarded by an international standards body. ISO 9000 certification can be quoted by an organization in official documents. However, SEI CMM assessment is purely for internal use.
- ♦ SEI CMM was specifically developed for software industry alone.
- ♦ SEI CMM goes beyond quality assurance and prepares on organization to ultimately achieve TQM. ISO 9000 aims at level 3 of SEI/CMM model

Chapter-8

Understanding the Computer Aided Software Engineering (CASE)

8.1 Briefly explain CASE benefits of CASE.
8.2 Briefly explain the building blocks for CASE
8.3 CASE support in software life cycle
8.4 List the different CASE tools.
8.1 Briefly explain CASE benefits of CASE.

8.1 Briefly Explain CASE Benefits of CASE

The term “Computer-Aided Software Engineering “can refer to the software used for the automated development of system software i.e. computer code. The CASE functions include analysis, design and programming. CASE tools automate methods for designing, documenting and producing structured computer code in the desired programming language. Many CASE tools are now available. Some CASE tools assist in phase-related takes such as specification, structured analysis, design , coding , testing etc. and some other CASE tools are related to non-phase activities such as project management and configuration management. The primary objectives of deploying CASE tools are

- To increase productivity
- To produce better quality software at lower cost

Two key ideas of computer-aided software system Engineering are :

- The harboring of computer assistance in software development and or software maintenance process.

- An engineering approach to the software development and or maintenance.

Some typical CASE tools are

- Configuration management tools
- Data modeling tools
- Model transformation tools
- Program transformation tools
- Source code generation tools
- Unified modeling language

CASE Environment

CASE tools are a class of software that automates many of the activities involved in various life cycle phases. Since CASE environments are classified based on the focus :

- Toolkits
- Language-centered
- Integrated
- Fourth generation
- Process-centered

Since different tools covering different stages share common information. It is required that they integrate through some central repository to have a consistent view of information associated with the software.

This central repository is usually a data dictionary containing the definitions of all composite and elementary data items. Through the central repository, all the CASE tools in a CASE environment share common information among themselves. Thus a CASE environment facilities the automation of the step-by-step methodologies for software development.

In contrast to a CASE environment a programming environment is an integrated collection of tools support only the coding phase of software development. The tools commonly integrated in a programming environment are a text editor, a compiler and a debugger. The different tools are integrated to the extent that once the compiler detects an error, the editor automatically goes to the statements in error and the error statements are highlighted. Examples of programming environments are Turbo C environment, Visual Basic, Visual c++ etc.

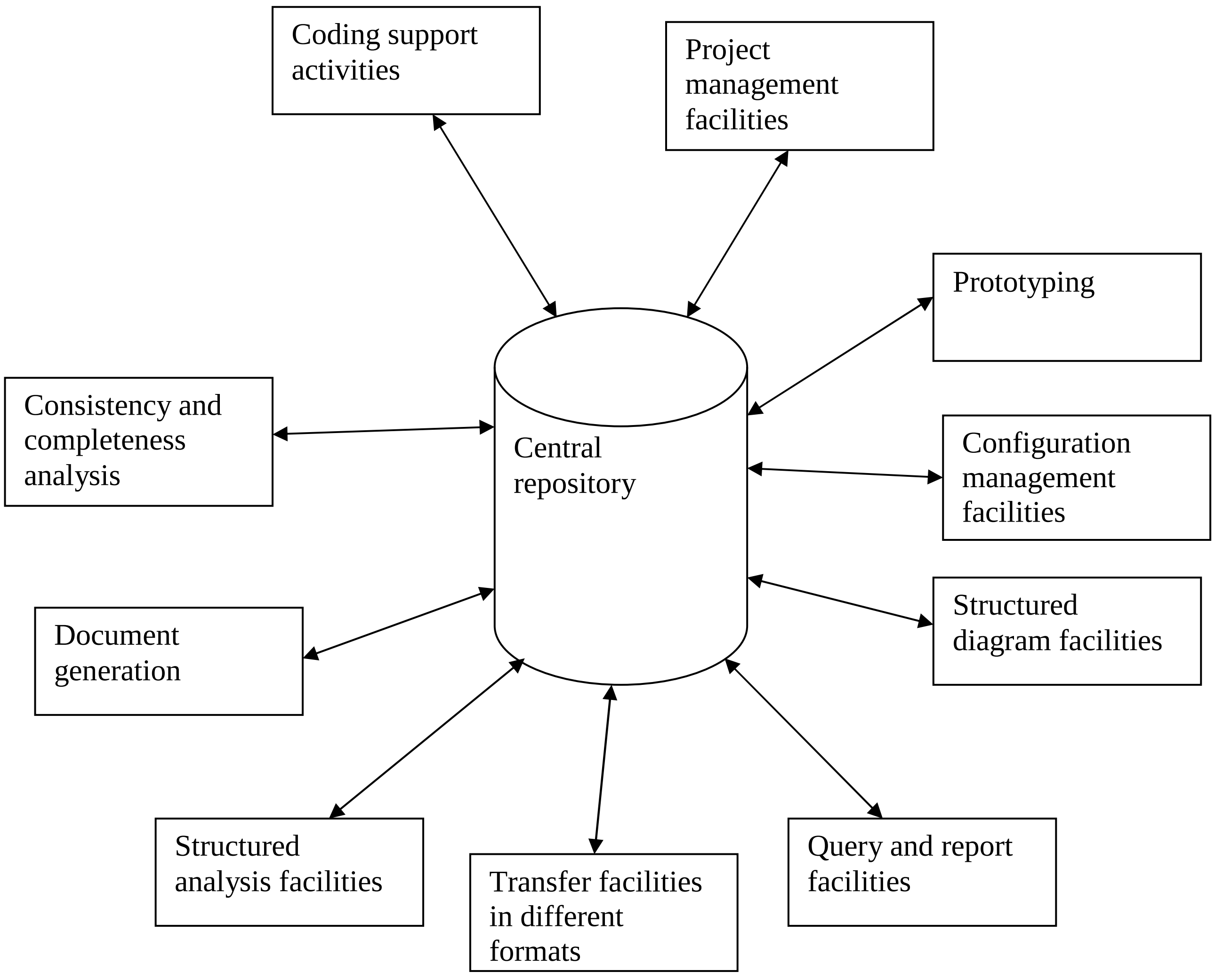

Coding support activities
Project management facilities
Prototyping
Consistency and completeness analysis
Central repository
Configuration management facilities
Structured diagram facilities
Document generation
Structured analysis facilities
Transfer facilities in different formats
Query and report facilities

Fig.8.1 A CASE environment

Benefits of CASE

- A key benefit arising use of a CASE environment is cost saving through all development phases.
- Use of CASE tools leads to considerable improvements to quality.
- CASE tools help produce high quality and consistent document since the important data relating to a software product are maintained in a central repository, redundancy in the stored data is reduced and therefore chances of inconsistence documentation are reduce to a great extent.
- CASE tools have led to drudgery in a software engineer's work.
- CASE tools have led to revolutionary cost savings in software maintenance efforts.
- Use of a CASE environment has an impact on the style of working of a company, and makes it conscious of structured and orderly approach.

8.2 Briefly Explain the Building Blocks for CASE

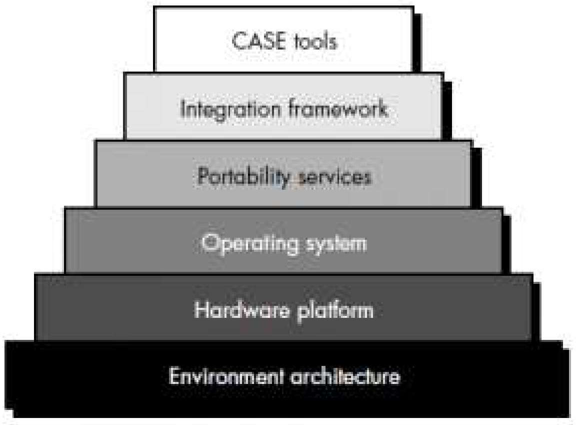


Figure 8.2 CASE building blocks

1) Environment Architecture: The environment architecture composed of the hardware platform and system support.

2) Hardware Platform
3) Operating System: Database and object management services.
4) Portability services: Allow CASE tools and their integration framework to migrate across different operating systems and hardware platforms without significant adaptive maintenance.
5) Integration framework: It is collection of specialized programs that allow CASE tools to communicate with one another .
6) CASE Tools : A CASE tool can be used quite effectively, even if it is a point solution.

8.3 CASE Support in Software Life Cycle

CASE tools should support a development methodology, help enforce the same and provide certain amount of consistency checking between different phases. The kind of support that CASE tools usually provide in the software development life cycle are :

a) Prototyping Support

The prototyping CASE tools requirements are

1. Define user interaction
2. Define the system control flow
3. Store and retrieve data required by the system
4. Incorporate some processing logic

There are several stand-alone prototyping tools. But a tools that integrates with the data dictionary can make use of entries in the data dictionary, help in populating the data dictionary and ensure the consistency between the design data and the prototype.

A good prototyping tool should support the following features :

- Since one of the main uses of a prototyping CASE tools is graphical user interface (GUI) development, a prototyping CASE tool should support the user to create a GUI using a graphics editor. The user should be allowed to define all data entry forms, menus and controls.

- It should integrate with the data dictionary of a CASE environment.
- If possible, it should be able to integrate with the external user-defined modules written in C or in some popular high level programming languages.
- The user should be able to define the sequence of states through which a created prototype can run. The user should also be allowed to control the running of the prototype.

b) Structured Analysis and Design

A CASE tool should support one or more of the structured analysis and design techniques. It should support effortlessly, making of the analysis and design diagrams. It should also support making of the fairly complex diagrams and preferably through a a hierarchy of levels. The CASE tools should provide easy navigation through different levels of design and analysis. The tools must support completeness and consistency checking across the design and analysis and through all levels of analysis hierarchy.

c) Code Generation

As for as code generation is concerned, the general expectation from a CASE tool is quite low. Pragmatic support expected from a CASE tools during code generation phase are :

- o The CASE tools should support generation of module skeletons or templates in one or more popular programming languages. It should be possible to include copyright message, brief description of the module, author name and the date of creation in some selectable format.

- The tools should generate records, structures, class, definitions automatically from the contents of the data dictionary in one or more popular programming languages.
- It should generate database tables for relational database management systems.

- The tools should generate code for user interface form prototype definitions for X-windows and MS Window-based applications.

d) Test CASE Generator

The CASE tool for test case generation should have the following features :

- It should support both design and requirement testing.
- It should generate test set reports in ASCII format which can be directly imported into the test plan document.

8.4 List the Different CASE Tools

♦ **Business process engineering tools :** Represent business data objects,
their relationships, and flow of the data objects between company business areas

♦ **Process modeling and management tools:** Represent key elements of processes and provide links to other tools that provide support to defined process activities.

♦ **Project planning tools:** Used for cost and effort estimation, and project scheduling .

- ♦ **Risk analysis tools:** Help project managers build risk tables by providing detailed guidance in the identification and analysis of risks.
- ♦ **Metrics and management tools:** Management oriented tools capture project specific metrics that provide an overall indication of productivity or quality, technically oriented metrics determine metrics that provide greater insight into the quality of design or code.
- ♦ **Documentation tools:** Provide opportunities for improved productivity by reducing the amount of time needed to produce work products
- ♦ **System software tools:** Network system software, object management services, distributed component support, and communications software.
- ♦ **Quality assurance tools:** Metrics tools that audit source code to determine compliance with language standards or tools that extract metrics to project the quality of software being built.
- ♦ **Database management tools:** RDMS and OODMS serve as the foundation for the establishment of the CASE repository
- ♦ **Analysis and design tools:** Enable the software engineer to create analysis and design models of the system to be built, perform consistency checking between models.
- ♦ **Prototyping tools:** Enable rapid definition of screen layouts, data design, and report generation.
- ♦ **Programming tools:** Compilers, editors, debuggers, OO programming environments, fourth generation languages, graphical programming environments, applications generators, and database query generators.
- ♦ **Integration and testing tools**
  - Data acquisition
    - get data for testing
  - Static measurement

    - analyze source code without using test cases
- Dynamic measurement
    - analyze source code during execution
- Simulation
    - simulate function of hardware and other externals)
- Test management
- Cross-functional tools

Reference Books

1. Fundamentals of Software Engineering

By

Rajib Mall

Prentice Hall of India

2. Software Engineering A Practitioner's Approach

By

Roger S. Pressman

McGraw-Hill International Edition

Model Question for Software Engineering

Model Question carrying 2 marks each.

1. What is a prototype?

2. What is project risk?

3. Define software reliability.

4.Differentiiate between verification and validation .

5. What do mean by debugging?

6.Distinguish between alpha and beta testing

7.What is Direct Manipulating Interface?

8.What do you mean by SRS ?

9.What is software reliability?

10. What is a structure chart?

11. What do you mean by CASE?

12.What is project planning?

13.What is staffing?

14. what is scheduling?

15. What is DFD?

16. Why should we use a life cycle model?

17Define object oriented concept.

18.Write down the structured analysis methodology.

19.Define coding standards and guidelines.

20. What is GUI?

21.What is function point metric?

22.Which software producted is treated as organic type?

23. Which software product is treated as embedded type?

24.What do you mean by coupling.

25. What is software engineering.

Model Question carrying 6 marks each

1. What is software reliability? Discuss the three software reliability metrics.

2. Describe how to get 9000 certification.
3.Explain Transform Analysis and Transaction Analysis.
4.What are the characteristics of good SRS document?
5.Discuss the project estimation technique.
6.Explain the main aspects of GUI.
7. Write down the rules for UID.
8.What is CASE tool? What are the benefits of CASE?
9.Differentiate between object oriented and function oriented software design?
10.Distinguish between cohesion and coupling. Classify cohesiveness.
11.Explain the features of spiral model.
12.Write down the effect of schedule change on cost.
13.Write down the work Breakdown Structure of scheduling.
14.Explain Activity networks of Scheduling.
15.Write down the concept of Gantt Chart & PERT Chart on scheduling.
16.Explain the software design approaches.
17.What is DFD? Write down the list of symbols used in DFD.
18. Explain code inspections.
19. Explain software documentation.
20. Explain debugging approaches & guidelines.
21. Explain the need for stress testing.
22. Explain error seeding of software testing.
23.Write down the importance of software reliability.
24.Explain reliability growth modelling
25. Write down the characteristics of quality software. Write down the evolution of software quality management system.
26.Briefly explain the building blocks for CASE.
27.Write down the limitations of DFD.
28. Explain code inspections methodology.
29.Explain software documentation.

30. Define system testing and explain various types of system testing approaches.

Model Question carrying 8 marks each.

1.What is cohesion and coupling? Explain the different types of cohesion and coupling.
2.Discuss the prototype model of software development.
3.Discuss about SEI Capability Maturity Model.
4. Explain UID Processs and models.
5. Explain interface design activities, defining interface objects, actions and the design issues.
6.Compare the various types of interface.
7.What is COCOMO model of estimation? Discuss the features of different COCOMO models.
8. What is cyclomatic complexity? Why it is used? Explain how cyclomatic complexity is computed? Give an illustration for this.
9. Explain the project estimation technique.
10. Explain the different phases of classical waterfall model.
11. Explain the different methods of white box testing techniques.
12.What is integration testing? Explain the different methods of integration testing.
13.Explain the steps of prototyping model with a diagram.
14.Write down the different steps of spiral model and explain.
15. Write down the responsibilities of a software project manager in software Engineering.
16.Explain organization structure with diagram.
17. Explain team structure with diagram.
18.Explain the classification of coupling.
19. Explain 0 level, 1 level, 2 level DFD with an example.
20.Write down the uses of structure chart & structured design.
21. Explain the principles of transformation of DFD to a structure chart.
22. Explain the different types user Interface so that the user can easily interact with the software.
23.Differentiate between object oriented and function oriented design approaches.
24. Explain different Black Box testing approaches used for software testing.
25.Explain the different metrics used for software size estimation.

26. Write short notes on:
    a. Spiral model
    b. FP based metric
    c. Jensen model for stating level estimation,
    d. project management.
    e. Black box testing
    f. Risk management