SRINIVAS UNIVERSITY

COLLEGE OF COMPUTER SCIENCE & TECHNOLOGY

CITY CAMPUS, PANDESHWAR,
MANGALORE-575 001

BACKGROUND STUDY MATERIAL

SOFTWARE ENGINEERING

B.C.A - V SEMESTER

Compiled by
Faculty

SOFTWARE ENGINEERING

CONTENTS

UNIT-1

2.4 Project Management Process

2.4.1 Phases of Management Process

2.4.2 Metrics, Measurements and Models

2.5 Software Configuration Management Process [SCM]

2.5.1 Configuration Identification

2.5.2 Change Control

2.5.3 Status Accounting and Auditing

2.6 Process Management Process

2.6.1 Building estimation models

2.6.2 Process Improvement and Maturity

Assignment 2

UNIT-2

3 . Software Requirement Analysis and Specification

3.1 Software Requirement

3.1.1 Need for SRS

3.1.2 Requirement Process

3.2 Problem Analysis

3.2.1 Analysis Issues

3.2.2 Informal Approach

3.2.3 Structured Analysis

3.2.4 Prototyping

3.3 Requirement Specification

3.3.1 Characteristics of an SRS

3.3.2 Components of SRS

3.3.3 Specification Languages

3.3.4 Structure of a requirements document

3.4 Requirement Validation

3.4.1 Requirement Reviews

UNIT 3

Project design management

4.1 Design Principles

4.1.1 Problem Partitioning and Hierarchy

4.1.2 Abstraction

4.2 Module-Level Concepts

4.2.1 Coupling

4.2.2 Cohesion

4.3 Design Notation and Specification

4.3.1 Structure Charts

4.3.2 Specifications

4.4 Structured Design Methodology

4.4.1 Restate the Problem as a Data Flow Diagram

4.4.2 Identify the Most Abstract Input and Output Data Elements

4.4.3 First- Level Factoring

4.4.4 Factoring the Input, Output and Transform Branches

4.4.5 Design Heuristics

4.5 Design Validation/Verification

4.5.1 Design Reviews

4.5.2 Automated Cross-Checking

Assignment 4

UNIT 4

5. Detailed Design and Programming Management

5.1 Module Specifications

5.1.1 Specifying Functional Modules

5.2 Detailed Design

5.2.1 PDL

5.2.2 Logic/Algorithm Design

5.3 Verification

5.3.1 Design Walkthroughs

5.3.2 Critical Design Review

5.3.3 Consistency Checkers

6. Coding

6.1 Programming Practice

6.1.1 Top-Down and Bottom-Up

6.1.2 Structured Programming

6.1.3 Information Hiding

6.1.4 Programming Style

6.1.5 Internal Documentation

6.2 Verification

6.2.1 Code Reading

6.2.2 Static Analysis

6.2.3 Symbolic Execution

6.2.4 Proving Correctness

6.2.5 Code Inspections or Reviews

6.2.6 Unit Testing

Assignment 6

UNIT 5

7. Testing and Maintenance

7.1 Introduction

7.2 Testing Fundamentals

7.3 Test Oracles

7.4 Top-Down and Bottom-up Approaches

7.5 Test Cases and Test Criteria

7.6 Psychology of Testing

7.7 Type of Testing

7.7.1 Functional Testing

7.7.2 Structural Testing

7.8 Levels of Testing

7.9 Test Plan

7.10 Software Maintenance Activities

7.11 Definitions of Software Maintenance

7.12 Types of Software Maintenance

7.12.1 Corrective Maintenance

7.12.2 Preventive Maintenance

7.12.3 Adaptive Maintenance

7.12.4 Perfective Maintenance

SOFTWARE ENGINEERING

Total hours:40H

UNIT I

8 hrs.

Introduction

Session 1: Definition of Software, the Software Problem.

Session 2: Software Engineering Approach.

Session 3: Phased Development Process, Software Process.

Session 4: Characteristics of the software Process, Process step Verification.

Session 5: Waterfall Model

Session 6: Iterative Enhancement Model, Spiral model.

Session 7: Process Management Process, SCM.

Session 8: Process Management Process,

UNIT – II

Software Requirements Analysis and Specifications

8 hrs.

Session 9: Software Requirements, Need for SRS,

Session 10: Requirement process

Session 11: Problem Analysis, Analysis Issues,

Session 12: Informal Approach, Structured Analysis

Session 13: Prototyping, Requirements Specification, Characteristics of an SRS

Session 14: Components of an SRS, Specification Languages

Session 15: Structure of a Requirements Document

Session 16: Validation, Requirement Reviews

UNIT – III

Project Design Management

8 hrs.

Session 17: Definition of Design and Design principles

Session 18: Problem partitioning and Hierarchy

Session 19 Abstraction and Module level Concepts.

Session 20: Coupling, Cohesion, Design Notation and Specification.

Session 21: Structure Charts, Specifications, Structured Design Methodology.

Session 22: Restate the Problem as a Data Flow Diagram, Identify the Most Abstract Input and Output Data Elements

Session 23: First- Level Factoring, Factoring the Input, Output and Transform Branches

Session 24: Design Heuristics, Design Validation/Verification, Design Reviews , Automated Cross-Checking

UNIT –IV

Detailed Design

8 hrs.

Session 25: Module specification, Specifying functional module, Detailed design, PDL

Session 26: Logic/Algorithm Design, Verification, Design Walkthroughs

Session 27: Critical Design Reviews, Consistency checkers

Session 28: Programming Practice, Top-Down and Bottom-Up

Session 29: Structured Programming, Information Hiding Programming Style, Internal Documentation

Session 30: Verification, Code Reading, Static Analysis

Session 31: Symbolic Execution and Execution Tree

Session 32: Proving Correctness, Code Inspections or Reviews, Unit Testing

UNIT – V

Testing and Maintenance

8 hrs.

Session 33: Testing Fundamentals, Error, Fault, and Failure, Test Oracles

Session 34: Top-Down and Bottom –Up Approaches, Test Cases and Test Criteria

Session 35: Psychology of Testing, Functional Testing

Session 36: Equivalence class partitioning, Boundary value analysis

Session 37: Cause-effect graphing, Structural Testing,

Session 38: Control flow based criteria, Data flow based testing

Session 39: Preventive and Corrective Maintenance

Session 40: Conclusion.

TEXT BOOKS

1. Integrated Approach to Software Engineering by Jalote Pankaj.

Scheme of Evaluation:
The paper carries 100 marks out of which 50 marks will be allotted to external examination and 50 marks will be allotted to the internal assessment.
Internal assessment marks will be calculated as follows

1. Performance in 2 IA examinations will be converted out of 30 marks
2. Attendance 10 marks
3. Assignment 10 marks.
   Total 50 marks.

External examination marks will be as follows

1.	1 marks questions 10 out of 12	1 X 10 = 10 marks.
2.	One full question out of 2 full questions in each unit carries	8 X 5 = 40 marks
		Total 50 marks.

In order to clear this paper minimum 50% marks must be scored both in internal and well as external examination.

****

EXAM:	**V SEM**	**PAPER:**	**18BCASD53**
SUBJECT:	**SOFTWARE ENGINEERING**	**CLASS:**	**BCA**
MAXIMUM:	**50**	**TIME:**	**2H**

WEIGHTAGE TO OBJECTIVES TABLE

SL.NO	OBJECTIVES	MARKS	% MARKS
1.	Knowledge (Remembering)	05	10
2.	Understanding	20	40
3.	Application	15	30
4.	Skill	10	20
Total		**50**	**100**

BLUE PRINT

EXAM:	**V SEM**	**PAPER:**	**18BCASD53**
SUBJECT:	**SOFTWARE ENGINEERING**	**CLASS:**	**BCA**
MAXIMUM:	**50**	**TIME:**	**2H**

Unit	REMEMBERING			UNDERSTAND			APPLICATION			SKILL			Total
	OT	SA	Unit wise Marks	OT	SA	Unit wise Marks	OT	SA	Unit wise Marks	OT	SA	Unit wise Marks	
1	1(1)	1(4)	5	1(1)	1(4)	5							10
2				2(1)	1(4)	6		1(4)	4				10
3					1(4)	4	2(1)	1(4)	6				10
4				1(1)	1(4)	5		1(4)	4	1(1)		1	10
5							1(1)		1	1(1)	2(4)	9	10
	05			20			15			10			50

UNIT-1
CHAPTER 1

INTRODUCTION TO S/W ENGINEERING

Introduction

The use of computers is growing very rapidly. Now computer systems are used in areas like business applications, scientific work, video games, aircraft control, missile control, hospital management, airline-reservation etc.

1.1 The Software Problem

Software is not only a collection of computer programs. There is a distinction between a *program* and *programming system's product*. A program is generally complete in itself and is used usually by the author of the program.

A programming system's product is used largely by people other than the developers of the system. The users may be from different backgrounds, so a proper user-interface should be provided. There is sufficient documentation to help the users to use the system.

IEEE defines **Software** as the collection of computer programs, procedures, rules and associated documentation and data. This definition clearly states that, software is not just programs, but includes all the associated documentation and data.

Note: IEEE stands for Institute of Electrical and Electronic Engineers

1.1.1 Software Is Expensive

The main reason for the high cost of the software is that, software development is still labor-intensive. In olden days, hardware was very costly. To purchase a computer lacks of rupees were required. Now a days hardware cost has been decreased dramatically. Now software can cost more than a million dollars, and can efficiently run on hardware that costs almost tens of thousands of dollars.

1.1.2. Late, Costly and Unreliable

There are many instances quoted about software projects that are behind the schedule and have heavy cost overruns. If the completion of a particular project is delayed by a year, the cost of the project may be double or still more. If the software is not completed in the scheduled period, then it will become very costly.

Unreliability means, the software does not do what it is supposed to do or does something it is not supposed to do. In software, failures occur due to bugs or errors that get introduced during the design and development process. Hence, even though the software may fail after operating correctly for sometime, the bug that causes the failure was there from the start. It only got executed at the time of failure.

1.1.3. Problem of Change and Rework

Once the software is delivered to the customer, it enters into maintenance phase. All systems need maintenance. Software needs to be maintained because there are often some residual errors remaining in the system that must be removed as they are discovered. These errors once discovered, need to be removed, leading to software getting changed. This is sometimes called as **corrective maintenance**.

Software often must be upgraded and enhanced to include more features and provide more services. This also requires modification of the software. If the operating environment of the software changes, then the software must be modified to the needs of the changed environment. The software must adapt some new qualities to fit to the new environment. The maintenance due to this is called **adaptive maintenance.**

1.2. Software Engineering Problem

Software Engineering is a systematic approach to the development, operation, maintenance and retirement of the software. There is another definition for s/w engineering, which states that "Software engineering is an application of science and mathematics by which the capabilities of computer equipments are made useful to man via computer programs, procedures and associated documentation".

1.2.1. Problem of Scale

A fundamental problem of software engineering is the problem of scale. Development of a very large system requires very different set of methods compared to developing a small system. In other words, the methods that are used for developing small systems generally do not scale up to large systems. For example: consider the problem of counting people in a room versus taking the census of a country. Both are counting problems but the methods used are totally different. A different set of methods have to be used for developing large software. Any large project involves the use of technology and project management. In small projects, informal methods for development and management can be used. However, for large projects both have to be much more formal. When dealing with small software project, the technology and project management requirement is low. However, when the scale changes to the larger systems, we have to follow formal methods. For example: if we have 50 bright programmers without formal management and development procedures and ask them to develop a large project, they will produce anything of no use.

1.2.2. Cost, Schedule and Quality

The cost of developing a system is the cost of resources used for the system, which in the case of software are, the manpower, hardware, software and other support resources. The manpower component is predominant as the software development is highly labor-intensive.

Schedule is an important factor in many projects. For some business systems, it is required to build a software with small cycle of time. The developing methods that produce high quality software is another fundamental goal of software engineering. We can view the quality of a software product having three dimensions: Product Operation, Product Transition and Product Revision.

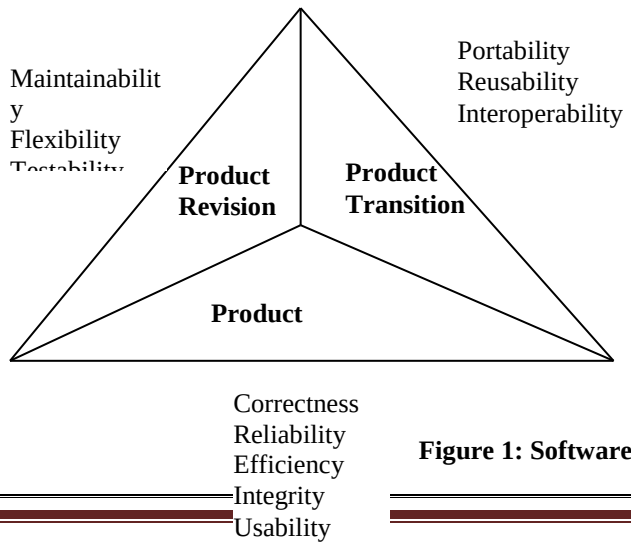


Figure 1: Software Quality Factors

The Product operation deals with the quality factors such as correctness reliability and efficiency. Product transition deals with quality factors such as portability, interoperability. Product revision deals with aspects related to modification of programs, including factors like maintainability and testability.

Correctness is the extent to which a program satisfies its specifications. **Reliability** is the property that defines how well the software meets its requirements. **Efficiency** is the factor in all issues rating to the execution of the software. It includes considerations such as response time, memory requirements and throughput. **Usability** is the effort required to learn and operate the software properly.

Maintainability is the effort required to locate and fix errors in the programs. **Testability** is the effort required to test and check that symbol or module performs correct operation or not. **Flexibility** is the effort required to modify an operational program (functionality).

Portability is the effort required to transfer the software from one hardware configuration to another. **Reusability** is the extent to which parts of software can be used in other related applications. **Inter-operability** is the effort required to couple the system with other systems.

1.2.3. The Problem of Consistency

For an organization there is another goal i.e. consistency. An organization involved in software development does not just want low cost and high quality for a project but it wants these consistently. Consistency of performance is an important factor for any organization; it allows an organization to predict the outcome of the project with reasonable accuracy and to improve its processes to produce higher-quality products. To achieve consistency, some standardized procedures must be followed.

1.3. Software Engineering Approach

The objectives of software engineering is to develop methods and procedures for software development that can scale-up for large systems and that can be used consistently to produce high quality software with low cost and small cycle time. The key objectives are high quality, low cost, small cycle time scalability and consistency. To achieve these objectives, design a proper software process and its control becomes the primary goal of software engineering. This process is called development process. The development process must be controlled properly. To do so we have project management which controls the development process to achieve the objectives.

1.3.1. Phased Development Process

A development process consists of various phases, each phase ending with a predefined output.

Software engineering must consist of these activities:

- Requirement specification for understanding and clearly stating the problem.
- Design for deciding a plan for the solution.
- Coding for implementing the planned solution.
- Testing for verifying the programs.

Requirement Analysis

Requirement analysis is done in order to understand the problem to be solved. In this phase, collect the requirement needed for the software project.

The goal of software requirement specification phase is to produce the **software requirement specification document**. The person who is responsible for requirement analysis is called as **analyst.** In problem analysis, the analyst has to understand the problem. Such analysis requires a thorough understanding of the existing system. This requires interaction with the client and end-users as well as studying the existing manuals and procedures. Once the problem is analyzed, the requirements must be specified in the requirement specification document.

Software Design

The purpose of design phase is to plan a solution for the problem specified by the requirement document. The output of this phase is the design document which is the blue-pint or plan for the solution and used later during implementation, testing and maintenance.

Design activity is divided into two phases- **System design** and **detailed design.** System design aims to identify the module that should be included in the system. During detailed design, the internal logic of each of the modules specified during system design is decided.

Coding

The goal of coding is to translate the design into code in a given programming language. The aim is to implement the design in the best possible manner. Testing and maintenance costs are much higher than the coding cost, therefore, the goal of should be to reduce testing and maintenance efforts. Hence the programs should be easy to read and understand.

Testing

After coding phase computer programs are available, which can be executed for testing purpose. Testing not only has to uncover errors introduced during coding, but also errors introduced during previous phases.

The starting point of testing is **unit testing.** Here each module is tested separately. After this, the module is integrated to form sub-systems and then to form the entire system. During integration of modules, **integration testing** is done to detect design errors. After the system is put together, **system testing** is performed. Here, the system is tested against the requirements to see whether all the requirements are met or not. Finally, **acceptance testing** is performed by giving user's real-world data to demonstrate to the user.

1.3.2. Project Management and Metrics

Development process does not specify how to allocate resources to different activities in a process. It also will not specify schedule for each activity, how to divide work within a phase, how to ensure that each phase is being done properly etc. Without properly handling these issues, it is unlikely that cost and quality objectives can be met. These types of issues are properly handled by project management. Software metrics are quantifiable measures that could be used to measure different characteristics of a software product.

LOC-Lines of code

DLOC-Delivered lines of code

KDLOC-Thousands of delivered lines of code

CHAPTER 2

SOFTWARE PROCESSES

2.1 Software Process

A process means, a particular method of doing something, generally involving several operations. In software engineering, the phrase software process refers to the method of developing the software. A software process is a set of activities together with proper ordering to build high-quality software with low cost and small cycle-time.

The process that deals with the technical and management issues of software development is called a software process. Clearly, many different types of activities need to be performed to develop software. As different types of activities are performed by different people, a software process consists of many components each consisting of many activities.

2.1.1 Processes, Projects and Products

A software process defines a method for developing software. A software project is a development project in which a software process is used. Software products are the outcomes of a software project. Each software development process starts with some needs and ends with some software that satisfies those needs. A software process specifies how the abstract set of activities that should be performed to go from user needs to final product.

The process specifies the activities at an abstract level that are not project specific. It is a generic set of activities and does not provide a detailed roadmap for a particular project. The process also specifies the order in which the activities of a project are carried out.

2.1.2. Components of Software Process

There are three major components in a software process. They are

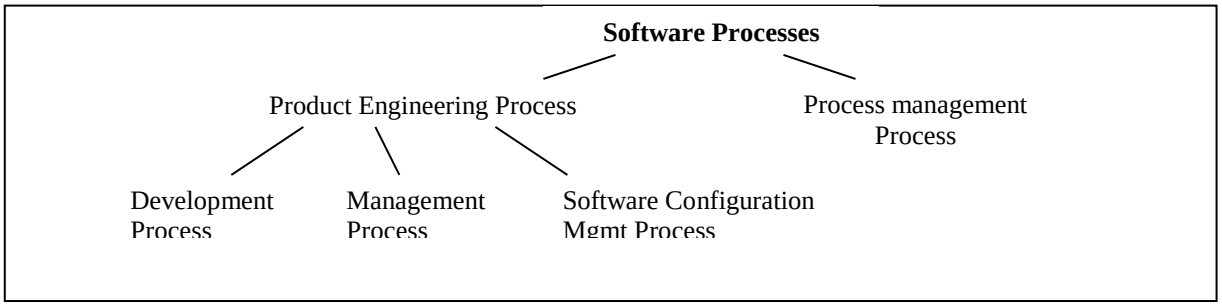
- Development process
- Management process
- Software configuration management process

The development process specifies the development and quality assurance activities that need to be performed. Management process specifies how to plan and control these activities, so that project objectives are met.

The development and management processes aim at satisfying cost and quality objectives of a project. Any software project has to deal with the problem of change and rework satisfactorily. This change cannot be handled by the development process. To handle the change and rework issues, another process called software configuration management process is used. The objectives of this process are to deal with managing changes so that cost and quality objectives are met and the integrity of the project is not violated due to the change requests.

Project management process and configuration control process depend on development process. The main objectives of these processes are to produce a desired product. So, they can be called as product engineering processes. Software is not static; it is dynamic because it must change to adapt newer technologies and tools. Due to this, a process to manage a software is needed. It is called process management process.

The main objectives of process management are to improve the software process. Improvement means that the capability of the process to produce high-quality s/w at low cost.

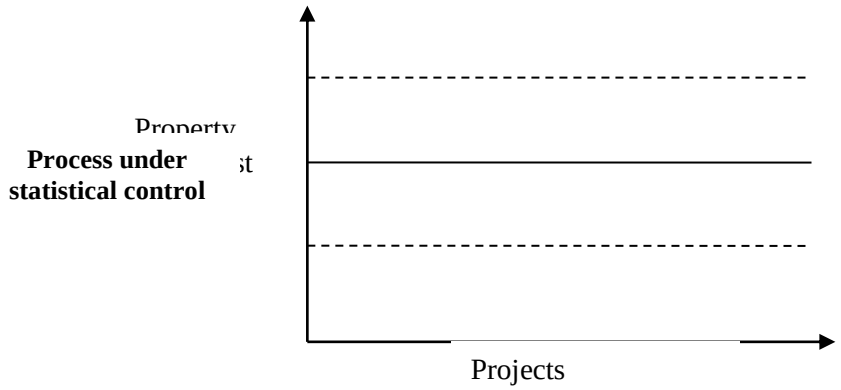


2.2. Characteristics of a Software Process

The fundamental objectives of software processes are optimality and scalability. Optimality means that the process must be able to produce high quality software at low cost and small cycle time. Scalability means that, it should also be applicable for large software projects. To achieve these objectives, the process must have some properties. Some characteristics of the software processes are listed below.

2.2.1. Predictability

Predictability of a process determines how accurately the outcome of following a process in a project can be predicted before the project is completed. Predictability is a fundamental property of any process. Effective management of quality assurance activities largely depend on the predictability of the process. A predictable process is also said to be under statistical control. A process is said to be under statistical control if following the same process produces similar results. Statistical control implies that most projects will be within a bound around the expected value. Any data beyond the line implies that the data and the project should be examined and followed to pass through only if a clear evidence is found that this is a **statistical aberration**.



2.2.2. Support Maintainability and Testability

Software products are not only easily maintainable because of the development process which is used for developing the software does not contain maintainability as a clear goal. Developers are made responsible for maintenance at least for a couple of years after developing the software.

Many examples show us that, programming is not a major activity where programmer spends his time. Testing consumes most resources during development. The goal of the process should not be to reduce the effort of design and coding but to reduce the effort of testing and maintenance. Both testing and maintenance depend heavily on design and coding and these costs are considerably reduced if the software is designed and coded make testing and maintenance easy.

2.2.3. Early Defect Removal and Defect Prevention

If there is a greater delay in detecting the errors, it becomes more expensive to correct them. As the figure given below shows, an error that occurs in the requirement phase if corrected during acceptance testing can cost about 100 times more than correcting the error in the requirement phase. To correct errors after coding, both the design and code are to be changed; there by changing the cost of correction. All the defect removal methods are limited in their capabilities and cannot detect all the errors that are introduced. Hence it is better to provide support for defect prevention.

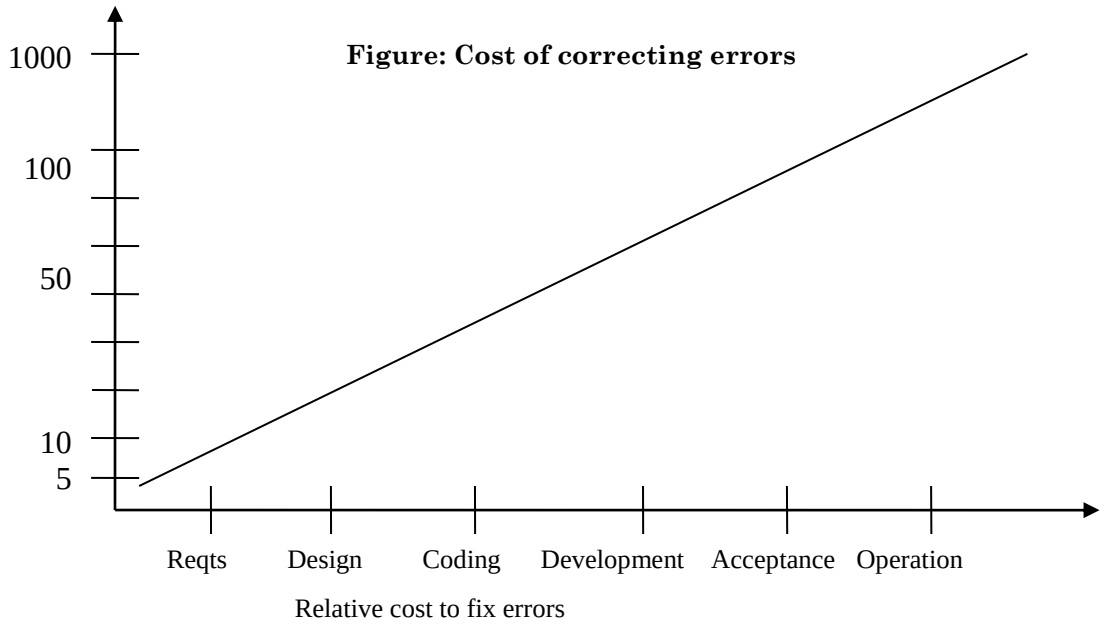


Figure: Cost of correcting errors

2.2.4. Process Improvement: Improving the quality and reducing the cost are the fundamental goals of the software engineering process. This requires the evaluation of the existing process and understanding the weakness in the process.

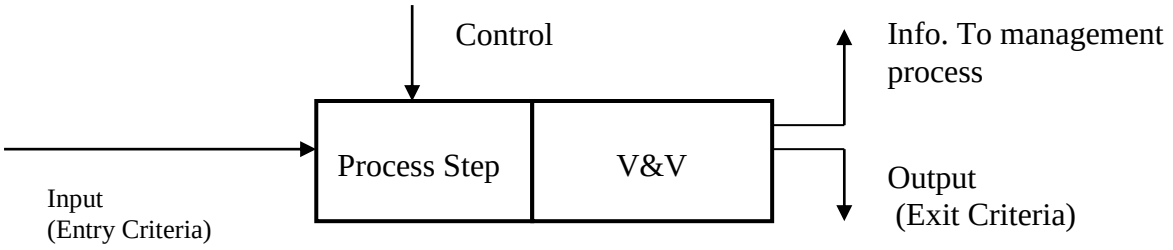
2.3. Software Development Process

Software development process focuses on the activities related to the production of the software. For example: design, coding, testing. A development process model specifies some activities that according to the model should be conducted. It also gives the order in which the activity to be performed.

2.3.1. A Process Step Specification

A production process is a sequence of steps. The output of one step will be the input to the next one. The process model will just specify the steps and their order. There are some practical issues such as when to initiate a step, when to terminate a step will not be given by any process model. There must be some verification and validation (V&V) at the end of each step in order to detect the defects. This implies that, there is an early defined output of a phase which can be verified by some means and can form the input to the next phase such products are called **work products.** [Example: Requirement document, design document, code prototype etc.] Having too many steps results in too many work products each requiring V&V can be very expensive. Due to this, the development process typically consists of a few steps producing few documents for V&V.

The major issues in development process are when to initiate and when to terminate a phase. This is done by specifying an entry criteria and exit criteria for a phase. The entry criteria specifies the conditions that the input to the phase should satisfy in order to initiate the activities of that phase. The output criteria specifies the conditions that the work product of current phase should satisfy in order to terminate the activities of the phase.



A step in a development process

Besides the entry and the exit criteria for the input and the output a development step needs to produce some information for the management process. To goal of management process is to control the development process.

2.3.2. Waterfall Model

Waterfall model is the simplest model which states that the phases are organized in a linear order. In this model, a project begins with feasibility analysis. On successfully demonstrating the feasibility of a project, the requirement analysis and project planning begins. The design starts after the requirement analysis is complete and the coding begins after the design is complete,

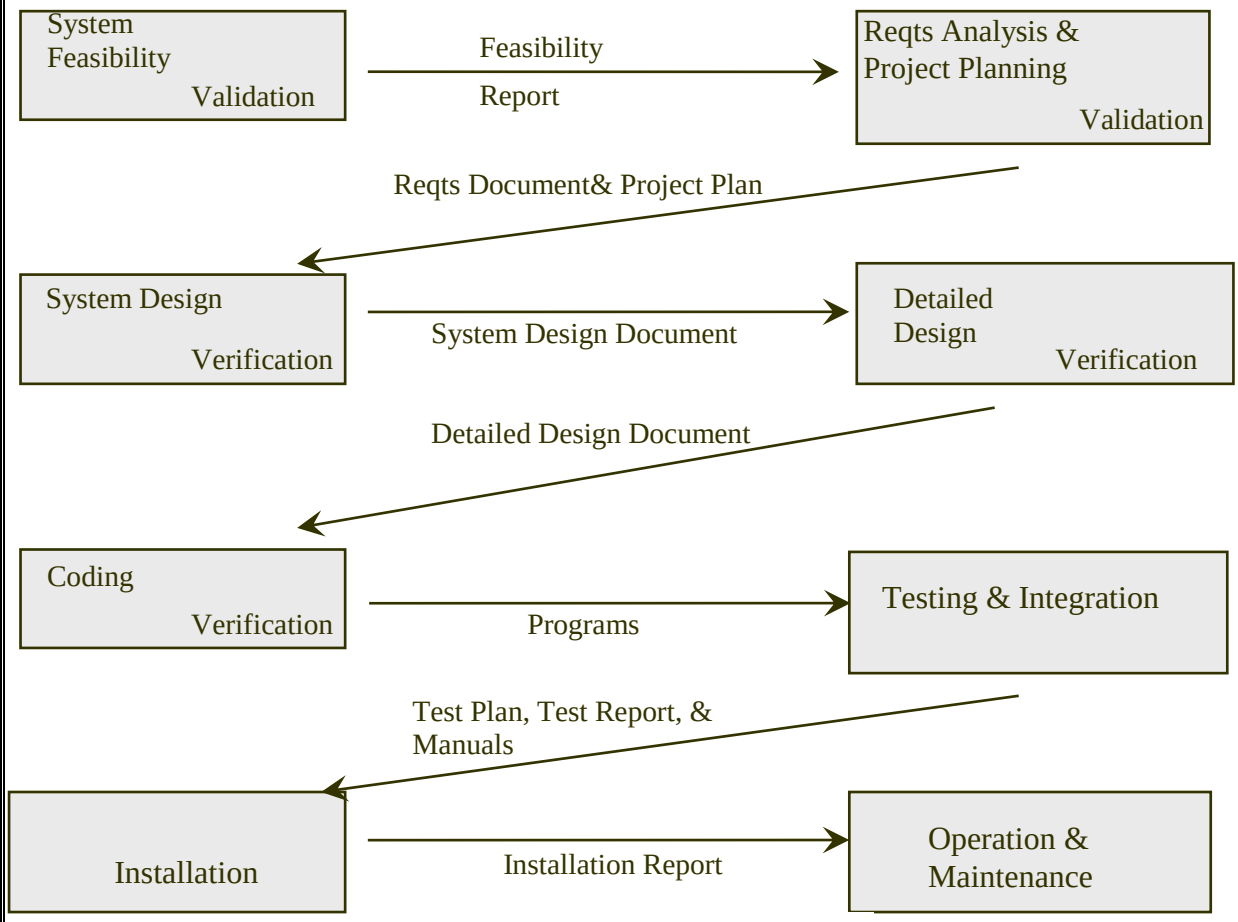
once the programming is complete, the code is integrated and testing is done. On successful completion of testing, the system is installed. After this, the regular operations and maintenance take place as shown in the figure (next page).

Each phase begins soon after the completion of the previous phase. Verification and validation activities are to be conducted to ensure that the output of a phase is consistent with the overall requirements of the system. At the end of every phase there will be an output. Outputs of earlier phases can be called as work products and they are in the form of documents like requirement document and design document. The output of the project is not just the final program along with the user manuals but also the requirement document, design document, project plan, test plan and test results.

Project Outputs of the Waterfall Model

- Requirement document
- Project plan
- System design document
- Detailed design document
- Test plan and test report
- Final code
- Software manuals
- Review report.

Reviews are formal meetings to uncover deficiencies in a product. The review reports are the outcomes of these reviews.



Waterfall Model

Limitations of Waterfall Model

1. Waterfall model assumes that requirements of a system can be frozen before the design begins. It is difficult to state all the requirements before starting a project.
2. Freezing the requirements usually requires choosing the hardware. A large project might take a few years to complete. If the hardware stated is selected early then due to the speed at which the hardware technology is changing, it will be very difficult to accommodate the technological changes.
3. Waterfall model stipulates that the requirements be completely specified before the rest of the development can proceed. In some situations, it might be desirable to produce a part of the system and then later enhance the system. This can't be done if waterfall model is used.
4. It is a document driven model which requires formal documents at the end of each phase. This approach is not suitable for interactive applications.
5. In an interesting analysis it is found that, the linear nature of the life cycle leads to "blocking states" in which some project team members have to wait for other team members to

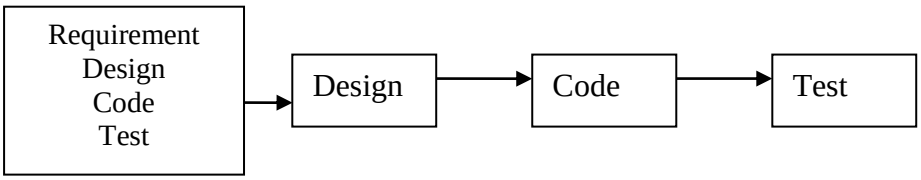
complete the dependent task. The time spent in waiting can exceed the time spent in productive work.

6. Client gets a feel about the software only at the end.

2.3.3. Prototype Model

The goal of prototyping is to overcome the limitations of waterfall model. Here a throwaway prototype is built to understand the requirements. This prototype is developed based on the currently known requirements. Development of the prototype undergoes design, coding and testing, but each of these phases is not done very thoroughly of formally. By using the prototype, the client can get actual feel of the system because the interaction with the prototype can enable the client to better understand the system. This results in more stable requirements that change less frequently. Prototyping is very much useful if there is no manual process or existing systems which help to determine the requirements.

Initially, primary version of the requirement specification document will be developed and the end-users and clients are allowed to use the prototype. Based on their experience with the prototype, they provide feedback to the developers regarding the prototype. They are allowed to suggest changes if any. Based on the feedback, the prototype is modified to incorporate the changes suggested. Again clients are allowed to use the prototype. This process is repeated until no further change is suggested.



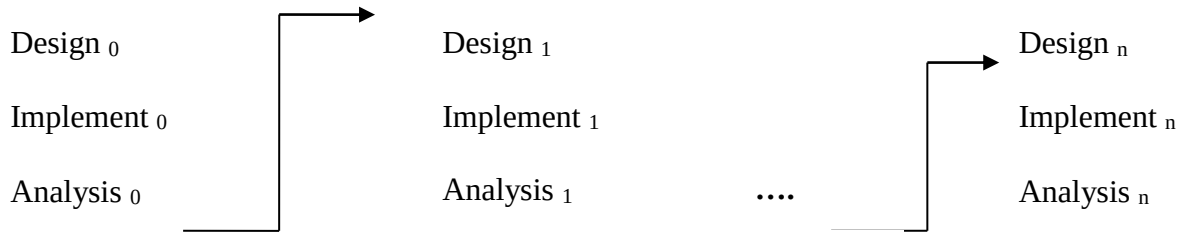
Requirement Analysis The Prototype Model

This model is helpful when the customer is not able to state all the requirements. Because the prototype is throwaway, only minimum documentation is needed during prototyping. For example design document and test plan etc. are not needed for the prototype.

Problems:

This model much depends on the efforts required to build and improve the prototype which in turn depends on computer aided prototyping tools. If the prototype is not efficient, too much effort will be put to design it.

2.3.4. Iterative Enhancement Model



Iterative Enhancement Model

This model tries to combine the benefits of both prototyping and waterfall model. The basic idea is, software should be developed in increments, and each increment adds some functional capability to the system. This process is continued until the full system is implemented. An advantage of this approach is that, it results in better testing because testing each increment is likely to be easier than testing the entire system. As prototyping, the increments provide feedback from the client, which will be useful for implementing the final system. It will be helpful for the client to state the final requirements.

Here a project control list is created. It contains all tasks to be performed to obtain the final implementation and the order in which each task is to be carried out. Each step consists of removing the next task from the list, designing, coding, testing and implementation and the analysis of the partial system obtained after the step and updating the list after analysis. These three phases are called design phase, implementation phase and analysis phase. The process is iterated until the project control list becomes empty. At this moment, the final implementation of the system will be available.

The first version contains some capability. Based on the feedback from the users and experience with the current version, a list of additional features is generated. And then more features are added to the next versions. This type of process model will be helpful only when the system development can be broken down into stages.

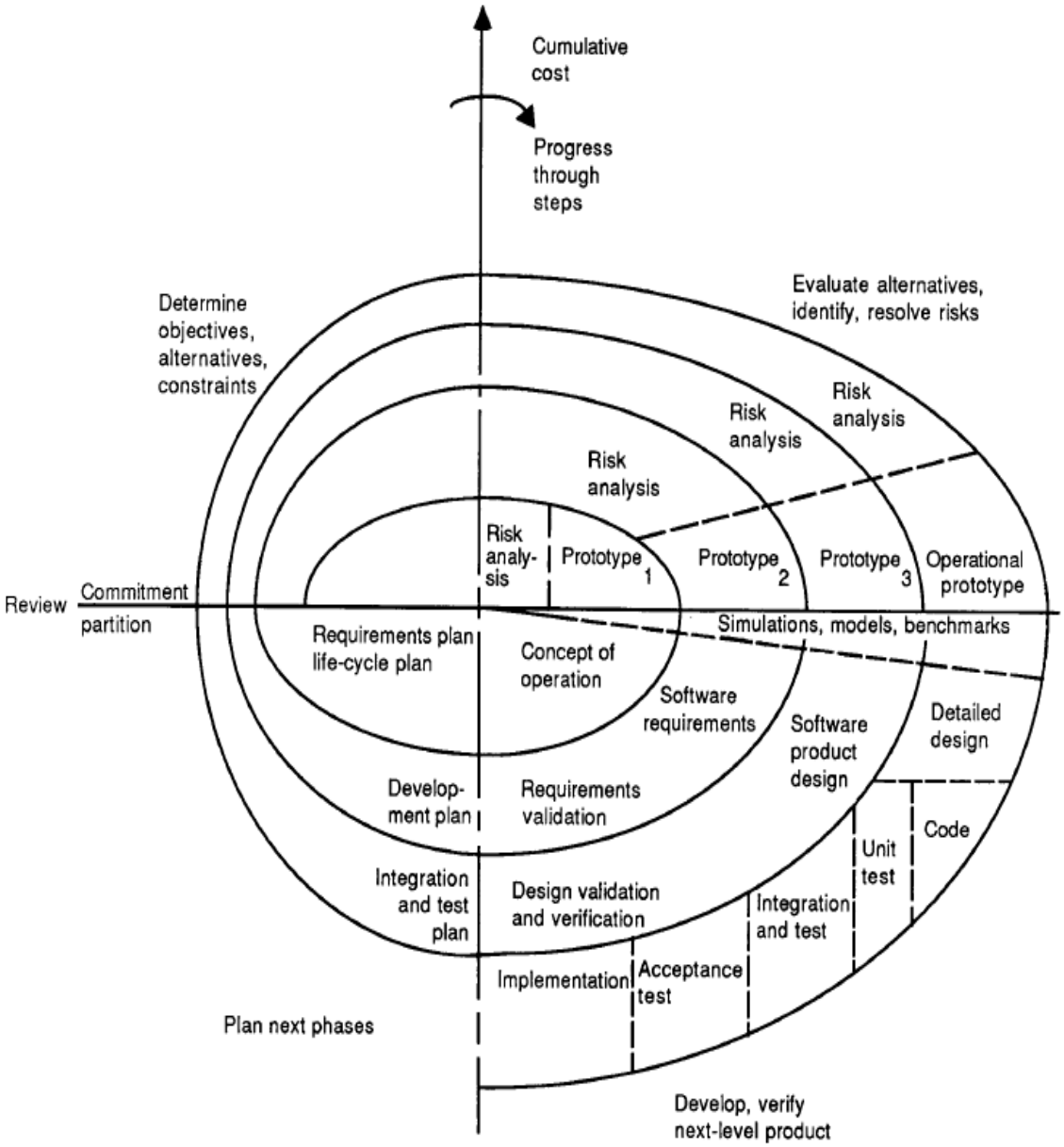
Disadvantage:

This approach will work only if successive increments can actually put into operation.

2.3.5. Spiral Model

As the name suggests, the activities of this model can be organized like a spiral that has many cycles as shown in the above figure. Each cycle in the spiral begins with the identification of objectives for that cycle; the different alternatives that are possible for achieving the objectives and the constraints that exist. This is the first quadrant of the cycle. The next step is to evaluate

different alternatives based on the objectives and constraints. The focus is based on the risks. Risks reflect the chances that some of the objectives of the project may not be met. Next step is to develop strategies that resolve the uncertainties and risks. This step may involve activities like prototyping.



The risk-driven nature of the spiral model allows it to suit for any applications. The important feature of spiral model is that, each cycle of spiral is completed by a review that covers all the products developed during that cycle; including plans for the next cycle. In a typical application of spiral model, one might start with an extra round-zero, in which the feasibility of the basic project objectives is studied. In round-one a concept of operation might be developed. The risks are typically whether or not the goals can be met within the constraints. In round-2, the top-

level requirements are developed. In succeeding rounds the actual development may be done. In a project, where risks are high, this model is preferable.

Problems:

1) It is difficult to convince the customers that the evolutionary approach is controllable.

2) It demands considerable risk-assessment expertise and depends heavily on this expertise for the success.

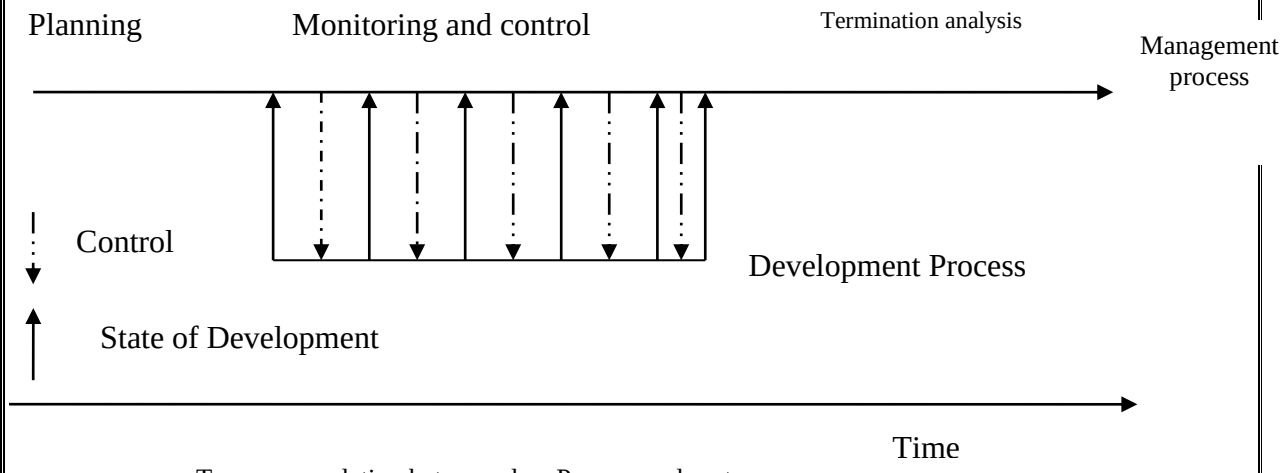
3) If major risks are uncovered and managed, major problems may occur.

2.4. Project Management Process

To meet cost, quality and schedule objectives, the resources have to be allocated for each activity of the project. The basic task of management process is to plan the detailed implementation of the process for the particular project and then ensure that the plan is followed. Proper management is essential for the success of any project.

2.4.1. Phases of Management Process

The activities of management process are grouped into three phases- planning, monitoring and control and termination analysis. Planning is the largest responsibility of the project management. The goal of this phase is to develop a plan for the software development. A software plan is usually produced before the development activities begin. The major activities during planning are cost estimation, schedule determination, project staffing, quality control etc.



Temporary relation between dev. Process and mgt. process

Project monitoring and control phase includes all activities that the project management has to perform while development is going on to ensure that project objectives are met and the development process proceeds according to the plan. If the objectives are not met, then this phase exerts suitable actions to control development activities. Monitoring requires proper information about the project. This information is obtained by the management process from the development process.

Termination analysis is performed when the development phase is over. The basic reason for performing termination analysis is to provide information about the development process. The ideal relationship between development process and management process is given in the figure above. It shows that, planning is done before starting the development process and termination analysis is conducted after the development is over. During development process, quantitative information flows to the monitoring and control phase of the management process which uses the information to exert control on the development process.

2.4.2. Metrics, Measurements and Models

Software metrics are quantifiable measures which could be used to measure the certain characteristics of the software. The quality of the software cannot be measured directly because software has no physical attributes. Metrics measurements and models go together. Metrics provide quantification of some property, measurement provides actual value for the metrics and models are needed to get the value for metrics that cannot be measured directly.

2.5. Software Configuration Management Process [SCM]

SCM is a process of identifying and defining the items in the system, controlling the change of these items throughout their life-cycle, recording and reporting the status of item and change request and verifying the completeness and correctness of these items. SCM is independent of development process. Development process handles normal changes such as change in code while the programmer is developing it or change in the requirement while the analyst is gathering the information. However it cannot handle changes like requirement changes while coding is being done. Approving the changes, evaluating the impact of change, decide what needs to be done to accommodate a change request etc. are the issues handled by SCM. SCM has beneficial effects on cost, schedule and quality of the product being developed.

It has three major components:

1. Software configuration identification
2. Change Control
3. Status accounting and auditing.

2.5.1. Configuration Identification:

When a change is done, it should be clear, *to what*, the change has been applied. This requires a **baseline** to be established. A baseline forms a reference point in the development of a system and is generally defined after the major phases in the development process. A software baseline represents the software in a most recent state. Some baselines are requirement baseline, design baseline and the product baseline or system baseline.

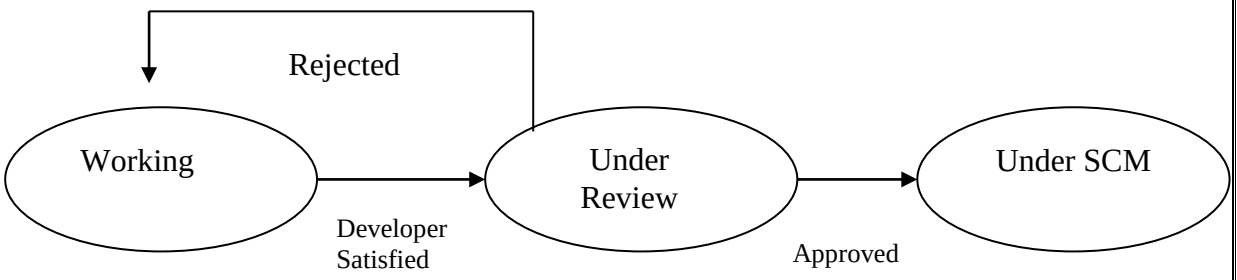
Though the goal of SCM is to control the establishment and changes to these baselines, treating each baseline as a single entity for the change is undesirable, because the change may be limited to a very small portion of the baseline. For this reason, a baseline can consist of many software configuration items. [SCI's] A baseline is a set of SCIs and their relations.

Because a baseline consists of SCIs and SCI is the basic unit for change control, the SCM process starts with identification of the configuration items. Once the SCI is identified, it is given a name and becomes the unit of change control.

2.5.2. Change Control:

Once the SCIs are identified, and their dependencies are understood, the change control procedures of SCM can be applied. The decisions regarding the change are generally taken by the configuration control board [CCB] headed by configuration manager [CM]

When a SCI is under development, it has considered being in working state. It is not under SCM and can be changed freely. Once the developer is satisfied with the SCI, then it is given to CM for review and the item enters to 'under review' state. The CM reviews the SCI and if it is approved, it enters into a library after which the item is formally under SCM. If the item is not approved, the item is given back to the developer. This cycle of a SCI is given in the figure below.



Once the SCI is in the library, it can not be modified, even without the permission of the CM. an SCI under SCM can be changed only if the change has been approved by the CM. A change is initiated by a change request (CR). The reason for change can be anything. The CM evaluates the CR primarily by considering the effect of change on the cost schedule and quality of the project and the benefits likely to come due to this change. Once the CR is accepted, project manager will take over the plan and then CR is implemented by the programmer.

2.5.3. Status Accounting and Auditing

The aim of status accounting is to answer the question like what is the status of the CR (approved/rejected), what is the average and effort for fixing a CR and what is the number of CR. For status accounting, the main source of information is CR. Auditing has a different role.

2.6 Process Management Process

In process management the focus is on improving the process that improves the general quality and productivity for the products produced using the process. One aspect of process management is to build models for the process.

2.6.1 Building Estimation Models

A model for the software process can be represented as

$Y=f(x_1,x_2,\ldots\ldots x_n)$.

The dependent variables y is the metric of interest (e.g. total effort, reliability etc). $x_1,x_2\ldots\ldots x_n$ are independent variables that represent some metric values that can be measured when this model is to be applied. The function f is really the model itself that specifies how y depends on these independent variables for the process.

The goal of building a model is to determine the relationship between a metric of interest and some other metric values. The simplest such model is one with only one independent variables, that is, to determine the relationship between two variables, say, y and x.

2.6.2 Process Improvement and Maturity

Process improvement requires understanding the current process and its deficiencies and then taking actions to remove the deficiencies. We present two frameworks that have been used by various organizations to improve their process.

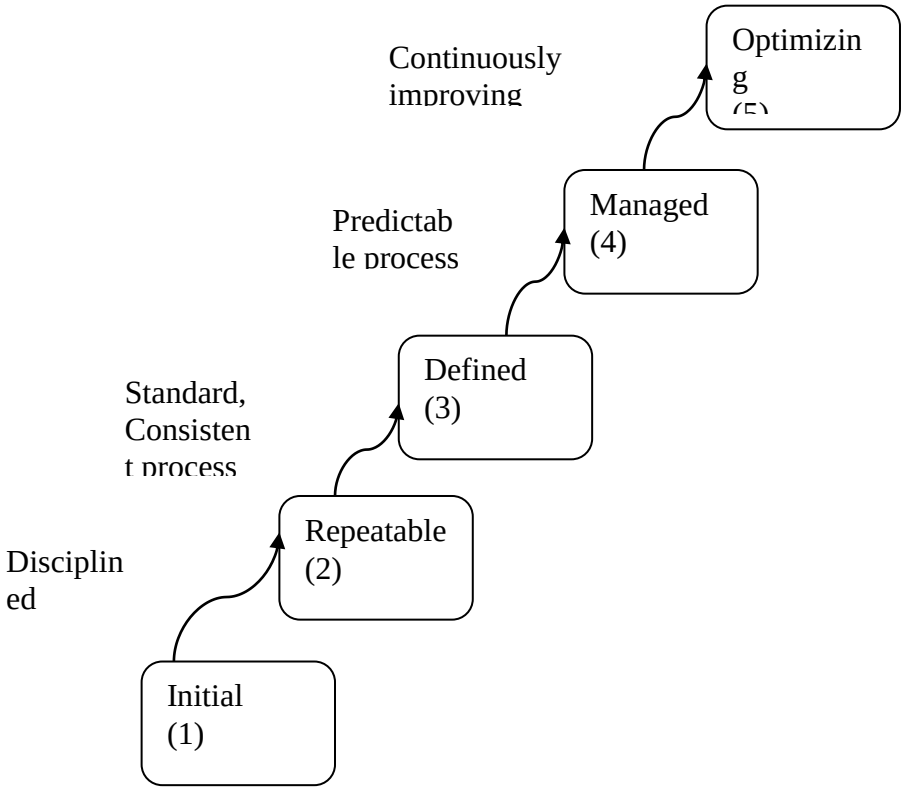
Capability Maturity Model

The goal of process improvement is to improve the process capability. The CMM suggests that there are five well defined maturity levels for a software process. These are initial(level1), repeatable, defined, managed and optimizing(level 5). The CMM framework says that as process improvement is best incorporated in small increments, processes go from their current levels to the next higher level when they are improved. During the course of process improvement, a process moves from level to level until it reaches level 5.

The initial process (level 1) is essentially an ad hoc process that has no formalized method for any activity. Organizations at this level can benefit most by improving project management, quality assurance, and change control.

In a repeatable process (level 2), policies for managing a software project and procedures to implement those policies exist. Ie,project management is well developed in a process at this level.

At the defined level (level 3), each step is carefully defined with verifiable entry and exist criteria, methodologies for performing the step, and verification mechanisms for the output of the step.



At the managed level (level 4) Data is collected from software processes, which is used to build models to characterize the process.

At the optimizing level (level 5), the focus of the organization is on continuous process improvement. Data is collected and routinely analyzed to identify areas that can be strengthened to improve quality or productivity. New technologies and tools are introduced and their effects measured in an effort to improve the performance of the process. Best software engineering and management practices are used throughout the organization.

Quality Improvement Paradigm and GQM

It gives a general method for improving a process, essentially implying that what constitutes improvement of a process depends on the organization to which the process belongs and its objectives. The basic idea behind this approach is to understand the current process, set objectives for improvement, and then plan and execute the improvement actions. The QIP consists of six basic steps:

- Characterize. Understand the current process and the environment it operates in.
- Set Goals. Based on the understanding of the process and the environment and objectives of the organization, set quantifiable goals for performance improvement.
- Choose Process. Based on the characterization and goals, choose the component processes that should be changed to meet the goals.
- Execute. Execute projects using the processes and provide feedback data.
- Analyze. Analyze the data at the end of each project.
- Package. Based on the experience gained from many projects, define and formalize the changes to be made to processes and expectation from the new processes.

Goal/Question/Metrics (GQM) paradigm suggests a general framework for collecting data from projects that can be used for a specific purpose. It is frequently used in the context of process improvement. The QIP and GQM have been used successfully by many organizations to improve their processes.

Part A (Multiple Choice Questions)
Remembering

1. IEEE defines _____ as the collection of computer programs, procedures, rules and associated documentation and data. Identify the same

A. SOFTWARE Engineering
B. **software**
C. software model
D. SRS

2. IEEE defines _____ is a systematic approach to the development, operation, maintenance and requirement of the software. Identify the same

A. **SOFTWARE Engineering**
B. requirement specification
C. coding
D. SRS

3. Quote from memory that the fundamental goal of software engineering is to produce _____

A. SRS
B. Review report
C. Cost effective design
D. <u>High quality software product</u>

4. Quote from the memory that _____ is the effort required to couple system with other system

A. Maintainability
B. **<u>Inter-operability</u>**
C. Portability
D. Quality

5. Quote from memory that _____ is the effort required to transfer the software from one hardware configuration to other.

A. Maintainability
B. Inter-operability
C. <u>Portability</u>
D. Reliability

6. State that _____ is the effort required to locate and fix the errors in the program.

A. <u>Maintainability</u>
B. Inter-operability
C. Portability
D. Testing

7. State that _____ is one of the quality attributes in product revision

A. Portability
B. Efficiency
C. Integrity
D. Testability

8. _____ is a risk driven model. Identify
A. Waterfall
B. **Spiral**
C. Iterative enhancement
D. Prototype

9. Blocking states is found in _____ model. Recognize.
A. W**aterfall**
B. Spiral
C. Iterative enhancement
D. Prototype

10. Freezing of requirement before development process is a limitation in _____, identify
A. **Waterfall**
B. Spiral
C. Iterative enhancement
D. Prototype

Understanding

11. Throw away models are used in _____model of development
A. Waterfall
B. Spiral
C. Iterative enhancement
D. Prototype

12. In which model , software is built in increments
A. Waterfall
B. Spiral
C. Iterative enhancement
D. Prototype

13. Evaluating a software process and identifying the loop holes increases _____ of a software
A. Predictability
B. Scalability
C. Cost
D. Quality

14. Development process of software has ____ phases
A. Three

B. Four
C. Five
D. Two

15. Software process components are development process, management process and _____process
A. Monitoring
B. SCM
C. Termination analysis
D. Quality analysis

16. Planning, monitoring and control and termination analysis are part of____
A. Development process
B. Quality analysis
C. SCM
D. Management process

17. _____provide quantifiable measures used to measure different characteristics of software product.
A. **Software Metric**
B. Software Model
C. Measurement
D. Mass

18. Expand SCM
A. Software Configuration Model
B. Software Code Management
C. **Software Configuration Management**
D. Software Code Maintenance

19. _____ is independent of development process
A. design
B. management
C. **SCM**
D. testing

20. Terms like Change Request, Status accounting appear in_____
A. management process
B. development process
C. quality analysis process
D. SCM process

21. Baseline in SCM process has set of ____
A. work products
B. Change Request (CR)
C. **software configuration Item**

D. Change Control Procedures

22. CMM, QIP, GQM are part of__
A. SCM process
B. Process improvement and maturity model
C. status accounting
D. estimation models

23. QIP has __basic steps.
A. four
B. five
C. **six**
D. Three

24. Initial, repeatable, defined, managed and optimizing are levels of___ model
A. QIP
B. GQM
C. **CMM**
D. SCI life cycle

25. In SCI life cycle, CR request is approved by____
a) **Configuration Manager**
b) CCB
c) Project manager
d) developer

Part B (4 marks)

Remember

1. Define Software Engineering. Explain various problem faced in Software Engineering
2. Discover the Quality attributes of Software Engineering
3. Discover different phases of development process
4. Discover in brief about the characteristics of a software process
5. State and explain the working of waterfall model with the help of a diagram.
6. List out the limitations of waterfall model

Understand

7. Explain the working of iterative enhancement model
8. Describe the spiral model with the help of diagram
9. Explain the phases involved in project management process
10. Describe the SCM life cycle of an item
11. Explain various activities of Software configuration Management Process
12. Identify the capability maturity Model

UNIT-2

CHAPTER-3

SOFTWARE REQUIREMENT ANALYSIS AND SPECIFICATION

3.1. Software Requirements

The software project is initiated by the client's needs. In the beginning, these needs are in the minds of various people in the client organization. The requirement analyst has to identify the requirements by talking to these people and understanding their needs. In the situation where the software is to automate the current manual process, many of the needs can be understood by observing the current practices. But no such methods exist in case of systems for which manual processes do not exist. (e.g., software for a missile control system) For such systems, the requirements may not be known even to the user. Requirements are to be visualized and created. Hence, identifying requirements necessarily involves specifying what some people have in their minds.

The inputs are to be gathered from different resources, these inputs may be inconsistent. The requirement phase translates the ideas in the minds of clients into a formal document. Software requirement specification (SRS) document is a document that completely describes 'what' the proposed software must do without describing how the software will do it. The basic goal of requirement phase is to produce SRS, which describes complete external behavior of the proposed software. There are several problems in gathering the requirements. All the requirements may not be known to any set of people. Another problem is changing requirements. Changing requirements is a continuous irritant for software developers and may lead to bitterness among the client and the developer. The final goal of the requirement phase is to produce a high quality and stable SRS.

3.1.1. Need for SRS

Client originates the requirements. The software is developed by software engineers and delivered to clients. Completed system will be used by the end-user. There are three major parties involved: client, developer and the end-user. The problem here is, the client usually does not understand software or the software development process and the developer often does not understand the client's problem and application area. This causes a communication gap between the client and the developer. A basic purpose of SRS is to bridge this communication gap. SRS is the medium with which the client and user needs are identified.

Another important purpose of developing the SRS is helping the clients to understand their own needs. In order to satisfy the client, he has to be made aware about the requirements of his organization. The process of developing an SRS helps here. Hence developing the SRS has many advantages as follows:

- **An SRS establishes the basis for agreement between the client and the developer on what the software product will do.**

This basis for agreement is frequently formalized into a legal contact between the client and the developer. So through SRS, the client clearly describes what is expected from the developer and the developer clearly understands the capabilities to build the software. Without such an agreement, it is almost guaranteed that once the development is over, the project will have an unhappy client and an unhappy developer.

- **SRS provides a reference for validation of the final product.**

SRS helps the client to determine if the software meets the requirements. Without proper SRS, there is no way for the client to determine if the software meets the requirements. Without a proper SRS there is no way a client can determine if the software being delivered is what was ordered and there is no way the developer can convince the client that all the requirements have been met.

- **A high-quality SRS is a prerequisite to high-quality software**.

The quality of the SRS has great impact on the cost of the project. The cost of fixing the error increases as we proceed on to the further phases in the software development life cycle. In order to reduce the errors the SRS should be of high quality.

- **A high-quality SRS reduces the development cost.**

If the SRS is of high quality, effort required to design the solution, effort required for coding, testing and maintenance will be reduced. Hence a high quality SRS reduces the development cost.

3.1.2. Requirement Process

The requirement process is the sequence of activities that need to be performed in the requirement phase. There are three basic activities in case of requirement analysis. They are:

1. Problem analysis or requirement analysis.
2. Requirement specification.
3. Requirement validation.

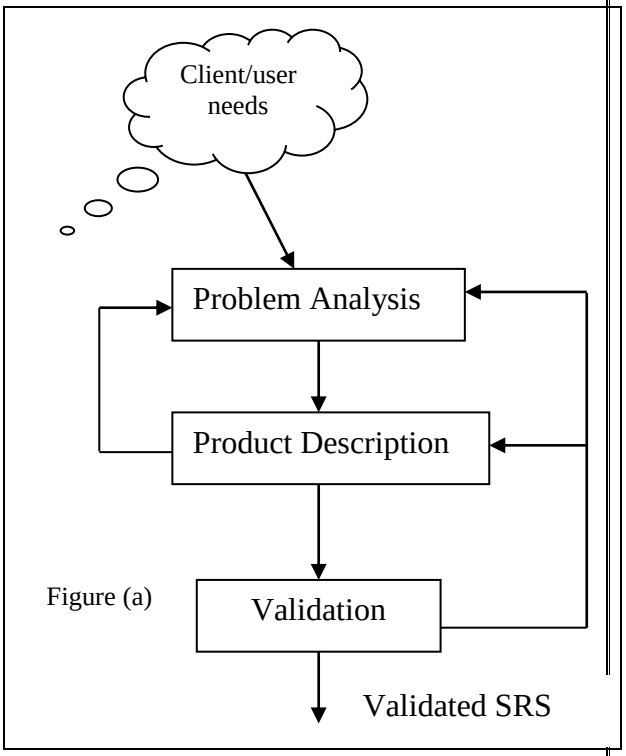


Figure (a)

Problem analysis is initiated with some general statement of needs. Client is the originator of these needs. During analysis, the system behavior, constraints on the system, its inputs, and outputs are analyzed. The basic purpose of this activity is to obtain the thorough understanding of what the software needs to provide. The requirement specification clearly specifies the requirements in the form of a document. The final activity focuses on validation of the collected requirements. Requirement process terminates with the production of the validated SRS.

Though it seems that the requirement process is a linear sequence of these activities, in reality it is not so. The reality is, there will be a considerable overlap and feedback between these activities. So, some parts of the system are analyzed and then specified while the analysis of some other parts is going on. If validation activities reveal some problem, for a part of the system, analysis and specifications are conducted again.

The requirement process is represented diagrammatically in figure (a). As shown in the figure, from specification activity we may go back to the analysis activity. This happens because the process specification is not possible without a clear understanding of the requirements. Once the specification is complete, it goes through the validation activity. This activity may reveal problems in the specification itself, which requires going back to the specification step, which in turn may reveal shortcomings in the understanding of the problem, which requires going back to the analysis activity.

During requirement analysis, the focus is on understanding the system and its requirements. For complex systems, this is the most difficult task. Hence the concept "divide-and-conquer" i.e., decomposing the problem into sub-problems and then understanding the parts and their relationship.

3.2. Problem Analysis

The basic aim of problem analysis is to obtain a clear understanding of the needs of the clients and users, what exactly is required from the software and what are the constraints on the analysis. Analysis involves interviewing the client and end users. These people and the existing documents about the current mode of operation are the basic source of information for the analysis. Typically, analysts research a problem by asking questions to the clients and end-users and by reading the existing documents.

Sometimes, client and end users do not understand or know all their needs. The analysts have to ensure that the real needs of the client are uncovered, even if they don't know them clearly. Here they act as consultants who play an active role of helping the clients and end users to identify their needs. Due to this, it is extremely important that the analyst thoroughly understands the client's organization.

3.2.1. Analysis Issues

During analysis, huge amount of information is collected in the form of answer to questions, questionnaires, studying the previous documents. It is important to have the complete set of requirements. Then the gathered information is to be organized so that it can be evaluated to check for completeness. Determining completeness requires active role of client during evaluation. Resolving contradictions that exist in the requirements gathered is another major job in case of requirement analysis. Contradictions occur because of lack of proper understanding about the system. Next major problem is avoiding internal design. Here, instead of concentrating on 'what's' the analysts may think about how to solve the problem. Such temptation should be restricted.

Information organization plays an important role in resolving the conflicts. During analysis, the interpersonal skills of the analyst are important rather than the technical skills. Interviewing the user and the client requires good communication skills. The situation can be more complex if

some users are reluctant to give the necessary information. Careful dealing is required to make sure that all that is required is found out.

To analyze the information some principles are to be known. The basic thing is: for complex tasks, divide and conquer method is used. That is, partition the problem into sub-problems and then try to understand each sub-problem and its relationship to other sub-problems in an effort to understand the whole problem. The question here is "partition with respect to what?" Generally, in analysis, the partition is done with respect to **object** or **function**. Most analysis techniques view the problem as consisting of objects or functions and aim to identify objects or functions and hierarchies and relationships among them.

An object is an entity in the real world that has clearly defined boundaries and independent existence. A function is a task, service or activity that is performed in the real world. The concept of state and projection can also be used to partition the process. A state of a system represents some conditions about the system. In each state, the expected behavior of the system is different. In projection, a system is defined in multiple point of view. There are three basic approaches to problem analysis: informal approach, structured analysis and prototyping.

3.2.2. Informal Approach:

The informal approach to analysis is the one where no defined methodology is used. Like in any approach, the information about the system is obtained by interaction with the client, end users, questionnaires, study of the existing documents, brainstorming etc. however, in this approach; no formal model is built of the system. In this approach, the analyst will have a series of meetings with the clients and the end users. In the early meetings, the clients and the end users will explain to the analyst about their work, their environment and their needs. Any documents describing the work may be given along with the outputs of the existing methods to perform tasks. In these meetings, analyst is basically a listener, absorbing the information provided. Once the analyst understands the system to some extent, he uses the next few meetings to seek clarifications of the parts he does not understand. He may document the information in some manner. In the final few meetings, the analyst essentially explains to the client about what he understands the system should do and uses the meetings as a means of verifying what he has gathered is true. An initial draft of SRS may be used in the final meetings.

3.2.3. Structured Analysis

The structured analysis technique uses function-based decomposition while modeling the problem. It focuses on the functions performed in the problem domain and the data consumed and produced by these functions. This method helps the analyst decide what type of information to obtain at different points in analysis, and it helps to organize information.

Data Flow Diagrams and Data Dictionary

Data flow diagrams (DFD) are commonly used during problem analysis. DFDs are quite general and are not limited to problem analysis. They were in use before software engineering discipline began. DFDs are very useful in understanding a system can be effectively used during analysis. DFD shows the flow of data through the system. It views a system as a function that transforms the input into desired output. Any complex system will not perform this in a single step and the data will typically undergo a series of transformations before it becomes an output. The DFD aims to capture the transformations that take place within a system to the input data so that eventually the output data is produced. The agent that performs the transformation of data from one state to another is called a process and is represented in the form of a circle (or bubble) in the DFD. The processes are shown by named circles and data flows are represented by named arrows entering or leaving the bubbles. Rectangles represent a source or sink and is a net originator or consumer of data. En example of DFD is given in figure given below.

This diagram represents the basic operations that are taking place while calculating the pay of employees in an organization. The source and sink both are worker here. Some conventions used in DFDs are: a labeled arrow represents an input or output. The need for multiple data flows by a process is represent by "*" between the data flows. This symbol represents AND relationship. For example, if "*" is there between two inputs A and B for a process, it means that A and B are needed for the process. Similarly the "OR" relationship is represented by a "+" between the data flows.

It should be pointed out that a DFD is not a flowchart. A DFD represents the flow of data, while a flow chart shows the flow of control. A DFD does not include procedural information. So while drawing a DFD, one must not get involved in procedural details and procedural thinking is consciously avoided. For examples, consideration of loops and decisions must be avoided. There are no detailed procedures that can be used to draw a DFD for a given problem. Only some directions can be provided. For large systems, it is necessary to decompose the DFD to further levels of abstraction. In such cases, DFDs can be hierarchically arranged.

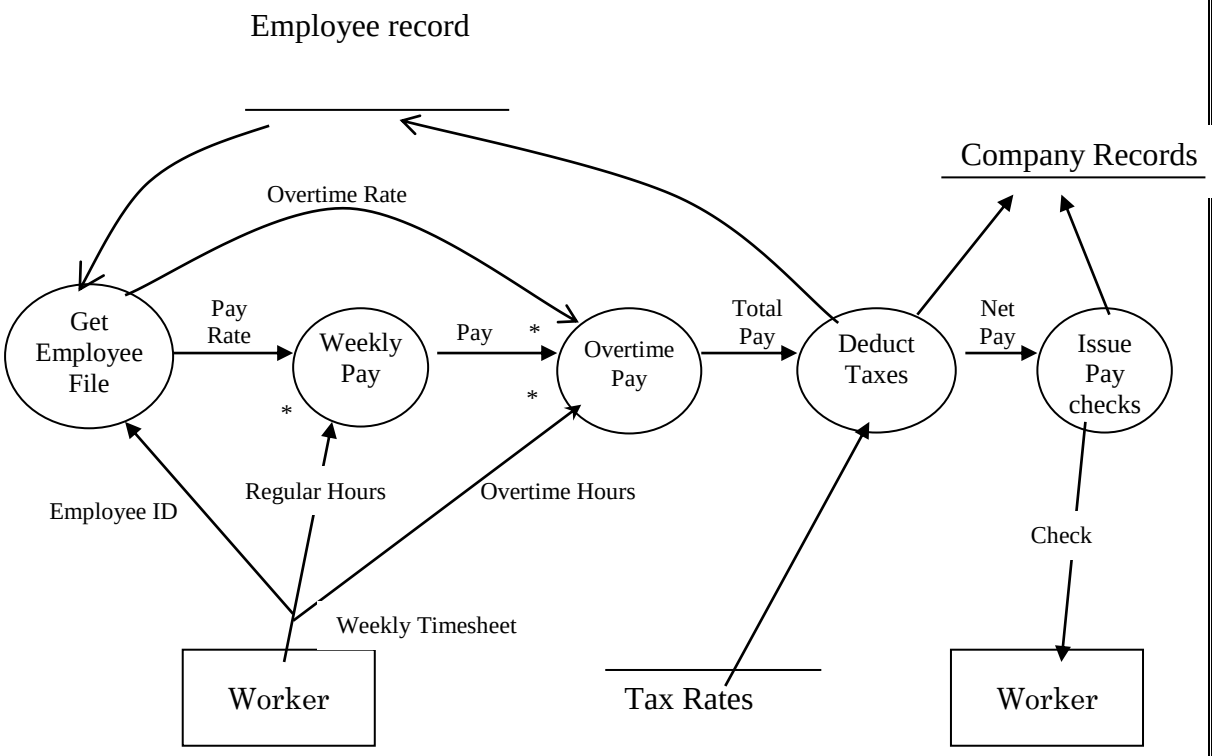


Figure: DFD of a system that pays workers

In a DFD, data flows are identified by unique names. These names are chosen so that they convey some meaning about what the data is. However, the precise structure of the data flows is not specified in a DFD. The data dictionary is a repository of various data flows defined in a DFD. Data dictionary states the structure of each data flow in the DFD. To define data structure, different notations are used. A composition is represented by +, selection is represented by / (i.e., either or relationship), and repetition may be represented by *. Example of a data dictionary is given below:

Weekly timesheet= employee_name +
employee_id+[regular_hrs+Overtime_hrs]*

Pay_rate= [hourly_pay/daily_pay /weekly_pay]

Employee_name= Last_name+ First_name +Middle_name

Employee_id= digit+ digit+ digit + digit

Most of the data flows in the DFD are specified here. Once we have constructed a DFD and associated data dictionary, we have to somehow verify that they are correct. There is no specific method to do so but data dictionary and DFDs are examined such that, the data stored in data dictionary should be there somewhere in the DFD and vice versa. Some common errors in DFDs are listed below:

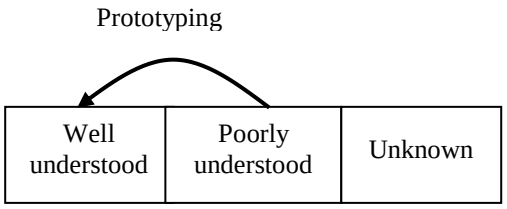
1. Unlabelled Data flows
2. Missing data flows (information required by a process is not available)

3. Extraneous data flows; some information is not being used in any process.
4. Consistency not maintained during refinement.
5. Missing Process
6. Contains some control information.

3.2.4. Prototyping

Prototyping is another method that can be used for problem analysis. It takes a very different approach to problem analysis as compared to structured analysis. In prototyping, a partial system is constructed, which is then used by the clients, developers and end users to gain a better understanding of the problem and the needs. There are two features to prototyping: **throwaway** and **evolutionary**. In the **throwaway approach**, the prototype is constructed with the idea that it will be discarded after the analysis is complete. In the evolutionary approach, the prototype is built with the idea that it will be used in the final system. Determining the missing requirements is an advantage of prototyping. In case of evolutionary prototyping, more formal techniques need to be applied since the prototype is retained in the final system.

Throwaway prototype leads to prototype model and the evolutionary prototype leads to iterative enhancement model. It is important to clearly understand when a prototype is to be used and when it is not to be used. The requirements can be divided into three sets — those that are well understood those that are poorly understood, and those that are unknown. In case of throwaway prototype, poorly understood ones that should be incorporated. Based on the experience with the prototype, these requirements then become well understood as shown in figure below.



It might be possible to divide the set of poorly understood requirements further into two sets—those critical to design and those not critical to design. If we are able to classify the requirements this way, throwaway prototype should focus on the critical requirements. There are different criteria to be considered when making a decision about whether or not to prototype. They are listed below.

Developer's application experience.

1. Maturity of application
2. Problem complexity.
3. Usefulness of early functionality
4. Frequency of changes
5. Magnitude of changes
6. Funds and staff
7. Access to users etc.

3.3. Requirement Specification

The final output of the requirement phase is the software requirement specification document. Lot of information will be collected in case of requirement analysis. SRS is written based on the knowledge acquired during analysis.

3.3.1. Characteristics of an SRS

A good SRS is:

1. Correct
2. Complete
3. Unambiguous
4. Verifiable.
5. Consistent
6. Ranked for important/stability
7. Modifiable
8. Traceable

1. Correct
2. Complete
3. Unambiguous
4. Verifiable.
5. Consistent
6. Ranked for important/stability
7. Modifiable
8. Traceable

A SRS is ***correc***t if every requirement included in SRS represents something required in the final system. An SRS is complete if everything the software is supposed to do and the responses of the software to all classes of input data are specified in the SRS. Completeness and correctness go hand-in-hand.

An SRS ***is unambiguous*** if and only if every requirement stated one and only one interpretation. Requirements are often written in natural language, which are inherently ambiguous. If the requirements are specified using natural language, the SRS writer should ensure that there is no ambiguity. One way to avoid ambiguity is to use some formal requirement specification language. The major disadvantage of using formal languages is large effort is needed to write an SRS and increased difficulty in understanding formally stated requirements especially by clients.

A SRS is ***verifiable*** if and only if every stored requirement is verifiable. A requirement is verifiable if there exists some cost effective process that can check whether the final software meets that requirement. Un-ambiguity is essential for verifiability. Verification of requirements is often done through reviews.

A SRS is ***consisten***t if there is no requirement that conflicts with another. This can be explained with the help of an example: suppose that there is a requirement stating that process A occurs before process B. But another requirement states that process B starts before process A. This is the situation of inconsistency. Inconsistencies in SRS can be a reflection of some major problems.

Generally, all the requirements for software need not be of equal importance. Some are critical. Others are important but not critical. An SRS is ***ranked for importance and/or stability*** if for each requirement the importance and the stability of the requirement are indicated. Stability of a requirement reflects the chances of it being changed. Writing SRS is an iterative process.

An SRS is ***modifiable*** if its structure and style are such that any necessary change can be made easily while preserving completeness and consistency. Presence of redundancy is a major difficulty to modifiability as it can easily lead to errors. For example, assume that a requirement is stated in two places and that requirement later need to be changed. If only one occurrence of the requirement is modified, the resulting SRS will be inconsistent.

An SRS is ***traceable*** if the origin of each requirement is clear and if it facilitates the referencing of each requirement in future development. Forward traceability means that each requirement should be traceable to some design and code elements. Backward traceability requires that it is possible to trace the design and code element to the requirements they support.

3.3.2. Components of SRS

The basic issues an SRS must address are:

1. Functional Requirements
2. Performance Requirements
3. Design constraints imposed on implementation
4. External interface requirements.

Functional Requirements

Functional requirements specify which output should be produced from the given inputs. They describe the relationship between the input and output of a system. All operations to be performed on the input data to obtain the output should be specified. This includes specifying the validity checks on the inputs and output data. Care must be taken not to specify any algorithm. An important part of the specification is, the system behavior in abnormal situations like invalid inputs or error during computation. The functional requirements must clearly state what the system should to if such situations occur. It should specify the behavior of the system for invalid inputs and invalid outputs. And also, the behavior of the system where the input is valid but normal operation cannot be performed should also be specified. An example of this situation is an airline reservation system, where the reservation cannot be made even for a valid passenger if the airplane is fully booked. In short, system behavior for all foreseen inputs and for all foreseen system states should be specified.

Performance Requirements

This part of the SRS specifies the performance constraints on the software system. There two types of performance requirements—***static*** and ***dynamic***. Static requirements do not impose constraints on the execution characteristics of the system. These include requirements like number of terminals to be supported, the number of simultaneous operations to be supported etc. These are also called capacity requirements of the system.

Dynamic requirements specify constraints on the execution behavior of the system. These typically include response time and throughput constraints on the system. All these requirements must be stated in measurable terms. Requirements like ":response time must be good " are not desirable because they are not verifiable.

Design Constraints

There are a number of factors in the client's environment that may restrict the choices of the designer. An SRS should identify and specify all such constraints.

Standard Compliance: This specifies the requirements for the standards the system must follow. The standards may include the report format and accounting procedures.

Hardware Limitations: the software may have to operate on some existing or pre-determined hardware, thus, imposing restrictions on the design. Hardware limitations can include the type of machines to be used, operating systems available, languages supported and limits on primary and secondary storage.

Reliability and Fault Tolerance: These requirements can place major constraints on how the system is to be designed. Fault tolerance requirements make the system more complex. Requirements in the system behavior in face of certain kinds of faults are to be specified. Recovery requirements deal with the system behavior in case of failure.

Security: These requirements place restriction on the use of certain commands, control access to data, provide different kinds of access requirements for different people etc.

External Interface Requirements

All the possible interactions of the software with the people, hardware and other software should be clearly specified. User interface should be user friendly. To create user friendly interface one can use GUI tools.

3.3.3. Specification Languages

Requirements can be verbally conveyed using the natural language. The use of natural language has some drawbacks. By the very nature of the natural language, written requirements will be imprecise and ambiguous. This goes against the desirable characteristic of the SRS. Due to these drawbacks, there is an effort to move from natural language to formal languages for requirement specification. In structured English, requirements are broken into sections and paragraphs; each paragraph is then broken into sub-paragraphs. The usage of words like “shall”, “perhaps”, “may be” etc are to be avoided.

Regular Expressions

Regular expressions can be used to specify the structure of stings. This specification is useful for specifying input data and content of the message. Regular expressions can be considered as grammar for specifying valid sequences in a language and can be automatically processed. They are routinely used in compiler construction. There are few basic constructs allowed in regular expressions:

1. Atoms: the basic symbol or alphabet of a language.
2. Composition: formed by concatenation two regular expressions. For regular expressions r1 and r2, concatenation is expressed as (r1 r2), and denotes the concatenation of strings represented by r1 and r2.
3. Alternation: Specifies the either/or relationship. For r1 and r2, the alternation is represented by (r1| r2) and denotes the union of the sets of strings specified by r1 and r2.
4. Closure: specifies the repeated occurrence of a regular expression. For a regular expression r, the closure is represented by r*, which means that the strings denoted by r are concatenated zero or more times.

Example: Consider a file containing student records.

Student_record=(Name Reg_no Courses)*
Name = (Last_name First_name)
Last_name, First_name= (A|B|C|D….. |Y|Z) (a|b|c|…….|y|z)*
Reg_no=digit digit digt digit digit digit
Digit=(0|1|2|……|9)
Courses=(C_number)*
C_number=(CS)(0|1|…)*

Decision Tables

Decision tables provide a mechanism for specifying complex decision logic. It is formal table-based notation that can be automatically processed to check for qualities like completeness and lack of ambiguity. A decision table has two parts. The top part lists different conditions and the bottom part specifies different actions. The table specifies under what combination of conditions what action is to be performed.

Example: Consider the part of a banking system responsible for debiting from the accounts. For this part the relevant conditions will be

C1: The account number is correct
C2: The signature matches
C3: There is enough money in the account.

The possible actions are

A1: Give money
A2: Give statement that not enough money is there in the account.
A3: Call the Police to check for fraud.

	1	2	3	4	5
C1	N	N	Y	Y	Y
C2		N	N	Y	Y
C3			N	Y	N
A1				X	
A2			X		X
A3		X			

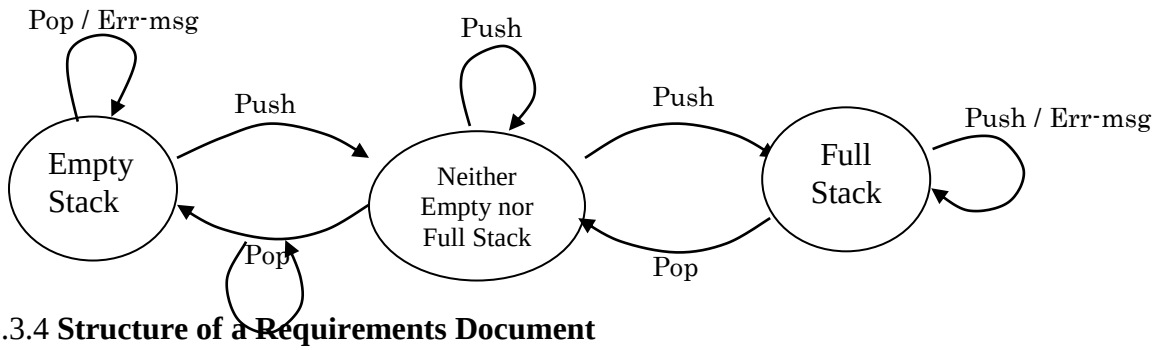
These conditions and possible actions can be represented in tabular form as follows:

The part of the decision table is shown here. For each condition a Y in the column indicates yes or true and N indicates no or false and a blank means that it can be either true or false. If an action is to be taken for a particular combination of the conditions, it is shown by X for that action. If there is no mark for an action for a particular combination of conditions, it means that the action is not to be performed.

Finite State Automata

FSA includes the concept of state and input data stream. It has a finite set of states and specifies transition s between the states. The transition from one state to another is based on the input.

Example: Consider that there is a stack. This stack has three different states— empty stack, neither empty nor full stack and full stack. Only two operations can be performed on stack they are push and pop. Then FSA for a stack can be as follows:



3.3.4 **Structure of a Requirements Document**

All the requirements for the system have to be included in a document that is clear and concise. For this, it is necessary to organize the requirements document as sections and subsections. There can be many ways to structure requirements documents.

The general structure of an SRS is given below.

1. Introduction

    1.1.1 Purpose

    1.1.2 Scope

    1.1.3 Definitions, Acronyms, and Abbreviations

    1.1.4 References

1.2 Overview

2. Overall Description

2.1 Product Perspective

2.2 Product Functions

2.3 User Characteristics

2.4 General Constraints

2.5 Assumptions and Dependencies

3. Specific Requirements

3.1 External Interface Requirements

3.1.1 User Interfaces

3.1.2 Hardware Interfaces

3.1.3 Software Interfaces

3.1.4 Communication Interfaces

3.2 Functional Requirements

3.2.1 Mode 1

3.2.1.1 Functional Requirement 1.1

.

.

3.2.1.n Functional Requirement 1.n

.

.

3.2.m Mode m

3.2.m.1 Functional Requirement m.1

3.3 Performance Requirements

3.4 Design Constraints

3.5 Attributes

3.6 Other Requirements

The introduction section contains the purpose, scope, overview, etc. of the requirements document. It also contains the references cited in the document and any definitions that are used. Section 2 describes the general factors that affects the product and its requirements. Product perspective is essentially the relationship of the product to other products. Defining if the product is independent or is a part of a larger product. A general abstract description of the functions to be performed by the product is given. Schematic diagrams showing a general view of different functions and their relationship with each other. Similarly, characteristics of the eventual end user and general constraints are also specified.

The specific requirements section describes all the details that the software developer needs to know for designing and developing the system. This is the largest and most important part of the documents. One method to organize the specific requirements is to first specify the external interfaces, followed by functional requirements, performance requirements, design constraints and system attributes.

The external interface requirements section specifies all the interfaces of the software: to people, other software, hardware, and other systems. User interfaces are clearly a very important component; they specify each human interface the system plans to have, including screen formats, contents of menus, and command structure. In hardware interfaces, the logical characteristics of each interface between the software and hardware on which the software can run are specified. In software interfaces, all other software that is needed for this software to run is specified, along with the interfaces. Communication interfaces need to be specified if the software communicates with other entities in other machines.

In the functional requirements section, the functional capabilities of the system are described. For each functional requirement, the required inputs, desired outputs, and processing requirements will have to be specified.

The performance section should specify both static and dynamic performance requirements.

The attributes section specifies some of the overall attributes that the system should have. Any requirement not covered under these is listed under other requirements. Design constraints specify all the constraints imposed on design.

3.4. Requirement Validation

The development of the software starts with the requirement document which is used to determine whether or not the delivered software is acceptable. It is important that the requirement specification contains no errors and specifies the client's requirements correctly. Due to the nature of requirement specification phase, there is a lot of room for misunderstanding and committing errors and it is quite possible that the requirement specification does not accurately represent the client's needs. The basic objective of requirement validation activity is to ensure that the SRS reflects the actual requirements accurately and clearly.

Common types of errors that occur during requirement phase are—**omission**, **inconsistency, incorrect fact and ambiguity.** Omission is a common error in requirements. In this type of error, some user requirements are simply not included in the SRS. Omission directly affects the completeness of SRS. Another error inconsistency can be due to contradictions within the requirements themselves or to incompatibility of the stated requirements with the actual requirements of the client. The third common error is incorrect fact. Errors of this type occur when some fact recorded in SRS is not correct. The fourth common error is ambiguity. These kinds of errors occur when there are some requirements that have multiple meanings.

3.4.1. Requirement Reviews

Requirement specification specifies the requirements of clients in a formal way. Hence the requirement validation also must include clients and end users. In requirement reviews, the SRS is carefully reviewed by a group of people. This group includes the author of the requirements document, someone who understands the needs of clients, a person of design team, and an independent software quality engineer. One way to organize the review meeting is to have each participant to go over the requirements before the meeting and mark the items he has doubts about or he feels need further clarification. Checklists can be quite useful in identifying such items. In meeting, each participant goes through the list of potential defects that he has uncovered. As members ask questions, the requirement analyst provides clarifications if there are no errors or agrees to the presence of errors. The discussions that take place during the reviews are likely to uncover the errors.

Alternatively, the meeting can take place with the analyst explaining each of the requirements in the document. The participants ask questions, share doubts or seek clarifications. Errors are uncovered by the participants. Here stress is given on detection of errors in the requirements and not to correct the errors. Correction is done later by the requirement analysts.

Part A(Multiple Choice Questions)
Understanding

1. The goal of requirement phase is to produce___

A. To produce System design document
B. To produce high quality and stable SRS
C. To produce Detailed design document
D. To produce review report

2. A requirement process involves ___ phases
A. four
B. three
C. five
D. two

3. The author of SRS document is____
A. developer
B. client
C. end user
D. **analyst**

4. Expand SRS
A. Software requirement structure
B. Structured requirements software
C. Software requirement specification
D. System requirement specification

5. An SRS establishes the basis for agreement between _____and_____on what the software product will do.

A. the Client and the Developer
B. the Client and the Designer
C. the User and the Analyst
D. the Client and the Analyst

6. DFD are used in____phase of requirement process
A. requirement review
B. requirement validation
C. structured problem analysis
D. informal problem analysis

7. _____ represents Flow of data in a system

A. DFD
B. Flow chart
C. Finite State Automata
D. SRS

8. In a DFD, a process is represented by _____
A. circle(bubble)
B. rectangle
C. source/sink
D. Oval

9. The system defined in multiple point of view is referred as
A. state
B. function
C. Object
D. **Projection**

10. Net originator or Consumer of data in DFD is represented by____
A. circle(bubble)
B. parallel lines
C. source/sink
D. named arrows

11. _____specifies the repeated occurrence of a regular expression

A. Composition
B. Closure
C. Alteration
D. Atoms

12. An Evolutionary prototype leads to _____model

A. Prototype model
B. Waterfall model
C. Iterative enhancement model
D. Spiral model

13. A SRS is _____, if requirement state only one interpretation.
A. consistent
B. complete
C. correct
D. **unambiguous**

14. if requirement stated doesn't conflict with another, then SRS is said to be___

A. Consistent
B. traceable
C. unambiguous
D. verifiable

15. Static and dynamic are the types observed in _____ requirement
A. functional
B. **performance**
C. design constraint
D. external user interface

16. Omission directly effects the ____ of SRS
A. Consistency
B. Completeness
C. traceability
D. modifiability

17. what is the error type ,if some requirements are not included in SRS
A. Inconsistency
B. Omission
C. Incorrect Fact
D. Ambiguity

18. _____requirement describe the relationship between the input and output of a system.
A. Performance requirement
B. Design constraints
C. External interface requirement
D. Functional requirement

19. Informal approach,structured analysis and _____ are the approaches to problem analysis
A. DFD
B. Finite state automata
C. Prototyping
D. Decision tables

20. Regular expressions, Finite state automata ,decision tables are part of
A. Prototyping
B. Specification language used for SRS
C. Structured analysis
D. Requirement review

21. Which symbol represents A source / sink in DFD
A. Circle

B. Rectangle
C. Parallel lines
D. Named Arrows

22. Redundancy is a major issue in____
A. Consistency
B. Completeness
C. traceability
D. modifiability

23. Which of the following is true about External interface requirement

A. It specifies the performance constraints on the software system
B. It specify which output should be produced from the given inputs
C. It specifies the requirements for the standards the system must follow
D. It specifies all the details of hardware,software support and other requirement to be stated

24. Which interface in SRS specifies the software communication with entities in the other machines
A. Hardware interface
B. Software interface
C. Communication interface
D. User interface

25. The _____ section in SRS document contains the purpose, scope, overview etc. of the requirements document

A. Introduction
B. Specific requirements
C. Functional requirements
D. Performance requirements

Part B (4 marks)
Understanding

1. Explain the need of SRS
2. Explain the phases of requirement process with diagram
3. Explain the role of analyst in problem Analysis
4. Describe the informal approach of problem analysis
5. Explain briefly structured analysis technique
6. Describe the prototyping technique and its types used for problem analysis

Application

7. Administer the role of DFD and data dictionary? Write different symbol with purpose used in DFD
8. Interpret the characteristics of SRS
9. Interpret various components of an SRS
10. Write the various factors considered in design constraint imposed on implementation
11. Write the general structure of SRS, explain in brief
12. Write a note on common errors occurred during requirement phase

UNIT 3

CHAPTER-4
FUNCTION ORIENTED DESIGN

Introduction

The design activity begins when the requirements document for the software to be developed is available. While the requirements specification activity is entirely in the problem domain, design is the first step in moving from the problem domain toward the solution domain. Design is essentially the bridge between requirements specification and the final solution for satisfying the requirements. The goal of the design process is to produce a model or representation of a system, which can be used later to build that system. The produced model is called the design of the system. The design of a system is essentially blueprint or a plan for a solution for the system. Here we consider a system to be a set of components with clearly defined behavior that interacts with each other in a fixed defined manner to produce some behavior or services for its environment.

The design process for software systems often has two levels. At the first level the focus is on deciding which modules are needed for the system, the specifications of these modules, and how the modules should be interconnected. This is what is called the system design or top-level design. In the second level, the internal design of the modules, or how the specifications of the module can be satisfied, is decided. This design level is often cal1ed detailed design or logic design. Detailed design essentially expands the system design to contain a more detailed description of the processing logic and data structures so that the design is sufficiently complete for coding. Because the detailed design is an extension of system design, the system design controls the major structural characteristics of the system. The system design has a major impact on the testability and modifiability of a system, and it impacts its efficiency. Much of the design effort for designing software is spent creating the system design.

A *design methodology* is a systematic approach to creating a design by applying of a set of techniques and guidelines. These techniques are not formalized and do not reduce the design activity to a sequence of steps that can be followed by the designer. The input to the design phase is the specifications for the system to be designed. Hence reasonable entry criteria can be that the

specifications are stable and have been approved, hoping that the approval mechanism will ensure that the specifications are complete, consistent, unambiguous, etc. The output of the top-level design phase is the architectural design or the system design for the software system to be built. This can be produced with or without using a design methodology. Reasonable exit criteria for the phase could be that the design has been verified against the input specifications and has been evaluated and approved for quality.

4.1. Design Principles

The design of a system is *correct* if a system built precisely according to the design satisfies the requirements of that system. Clearly, the goal during the design phase to-produce correct designs. However, correctness is not the sole criterion during the design phase, as there can be many correct designs. The goal of the design process is not simply to produce *a* design for the system. Instead, the goal is to find the *best* possible design within the limitations imposed by requirements and the physical and social environment in which the system will operate. A design should clearly be verifiable, complete (implements all the specifications), and traceable (all design elements can be traced to some requirements).

However, the two most important properties that concern designers are efficiency and simplicity. *Efficiency* of any system is concerned with the proper use of scarce resources by the system. The need for efficiency arises due to cost considerations. If some resources are scarce and expensive, it is desirable that those resources be used efficiently. In computer systems, the resources that are most often considered for efficiency are processor time and memory. An efficient system is one that consumes less processor time and requires less memory.

Simplicity is perhaps the most important quality criteria for software systems. We have seen that maintenance of software is usually quite expensive. Maintainability of software is one of the goals we have established. The design of a system is one of the most important factors affecting the maintainability of a system. During maintenance, the first step a maintainer has to undertake is to understand the system to be maintained. Only after a maintainer has a thorough

understanding of the different modules of the system, how they are interconnected, and how modifying lone will affect the others should the modification be undertaken.

Creating a simple (and efficient) design of a large system can .be an extremely complex task that requires good engineering judgment. As designing is fundamentally a creative activity, it cannot be reduced to a series of steps that can be simply followed, though guidelines can be provided. In this section we will examine some basic guiding principles that can be used to produce the design of a system. Some of these design principles are concerned with providing means to effectively handle the complexity of the design process. Effectively handling the complexity will not only reduce the effort needed for design (i.e., reduce the design cost), but can also reduce the scope of introducing errors during design.. In fact, the methods are also similar because in both analysis and design we are essentially constructing models.

There are some fundamental differences between the design and the problem analysis phase. First, in problem analysis, we are constructing a model of the problem domain, while in design we are constructing a model for the solution domain. Second, in problem analysis, the analyst has limited degrees of freedom in selecting the models as the problem is given, and modeling has to represent it. In design, the designer has a great deal of freedom in deciding the models, as the system the designer is modeling does not exist; in fact the designer is creating a model for the system that will be the basis of building the system. That is, in design, the system depends on the model, while in problem analysis the model depends on the system. Finally, as pointed out earlier, the basic aim of modeling in problem analysis is to understand, while the basic aim of modeling in design is to optimize (in our case, simplicity and performance)

4.1.1. Problem Partitioning and Hierarchy

When solving a small problem, the entire problem can be tackled at once.. For solving larger problems, the basic principle is "divide and conquer." Clearly, dividing in such a manner that all the divisions have to be conquered together is riot the intent of this wisdom. "For software design, therefore, the goal is to divide the problem into manageably small pieces that can be solved separately. However, the different pieces cannot be entirely independent of each other, as together form the system. The different pieces have to cooperate and communicate to solve the

larger problem. This communication adds complexity, which arises due to partitioning and may not have existed in the original problem. As the number of components increases, the cost of partitioning, together with the cost of this added complexity, may become more than the savings achieved by partitioning. The designer has to make the judgment about when to stop partitioning. Clearly proper partition minimizes the maintenance cost.

The two of the most important quality criteria for software design are simplicity and understandability. It can be argued that maintenance is minimized if each part in the system can be easily related to the application and each piece can be modified separately. If a piece can be modified separately, we call it *independent* of other pieces. If module A is independent of module B, then we can modify A without introducing any unanticipated side effects in B. Total independence of modules of one system is not possible, but the design process should support as much independence as possible between modules. Dependence between modules in a software system is one of the reasons for high maintenance costs. Clearly, proper partitioning will make the system easier to maintain by making the design easier to understand.

4.1.2. Abstraction

An abstraction of a component describes the external behavior of that component without bothering with the internal details that produce the behavior. The abstract definition of a component is much simpler than the component itself. Abstraction is a crucial part of the design process and is essential for problem partitioning. Partitioning essentially is the exercise in determining the components of a system. However, these components are not isolated from each other; they interact with each other, and the designer has to specify how a component interacts with other components. To decide how a component interacts with other components, the designer has to know, at the very least, the external behavior of other components. To allow the designer to concentrate on one component at a time, abstraction of other components is used. Abstraction is used for existing components as well as components that are being designed. Abstraction of existing components plays an important role in the maintenance phase. Using these abstractions, the behavior of the entire system can be understood. This also helps determine how modifying a component affects the system. During design, the components do not exist, and in the design the designer specifies only the abstract specifications of the different components.

The basic goal of system design is to specify the modules in a system and their abstractions. Once the different modules .are specified, during the detailed design the designer can concentrate on one module at a time. The task in detailed design and implementation is essentially to implement the modules so that the abstract specifications of each module are satisfied.

There are two common abstraction mechanisms for software systems: *functional abstraction* and *data abstraction.* In *functional abstraction,* a module is specified by the function it performs. For example, a module to compute the log of a value can be abstractly represented by the function log. Similarly, a module to sort an input array can be represented by the specification of sorting. Functional abstraction is the basis of partitioning in function-oriented approaches. That is, when the problem is being partitioned, the overall transformation function for the system is partitioned into smaller functions that comprise the system function. The second unit for abstraction is *data abstraction.* Any entity in the real world provides some services to the environment to which it belongs. Often the entities provide some fixed predefined services. Data abstraction supports this view. Data is not simply a collection of objects but is treated as objects with some predefined operations. It is possible top view this object at an abstract level. From outside an object, the internals of the object – are hidden. Only the operations on the object are visible. Data abstraction forms the basis fop *object-oriented design.*

Modularity

As mentioned earlier, the real power of partitioning comes if a system is partitioned into modules so that the modules are solvable and modifiable separately. It will be even better if the modules are also separately compliable. A system is m*odular* if it consists of discreet components so that each component can implemented separately, and a change to one component has minimal impact on the other components. The modularity is a clearly a desirable property in a system. Modularity helps in system debugging. Software system cannot be made modular by simply chopping it into a set of modules. For modularity, each module needs, to support a well-defined abstraction and a clear interface through which it can interact with other modules.

Top-Down and Bottom-Up Strategies

A system consists of components; a System is a hierarchy of components. The highest level component corresponds to the total system. To design such a hierarchy there is two possible approaches: top-down and bottom-up. The top-down approach starts from the highest-level component of the hierarchy and proceeds through to lower levels. By contrast, a bottom-up approach starts with the lowest-level component of the hierarchy and proceeds through progressively higher levels to the top-level component. A top-down design approach starts by identifying the major components of the system, decomposing them into their lower-level components and iterating until the desired level of detail is achieved. Top-down design methods often result in *stepwise refinement.* Starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly.

A bottom-up design approach starts with designing the most basic or primitive components and proceeds to higher-level components that use these lower-level components. Bottom-up methods work with *layers of abstraction.* Starting from the very bottom, operations that provide a layer of abstraction are implemented. The operations of this layer are then used to implement more powerful operations and a still higher layer of abstraction, until the stage is reached where the operations supported by the layer are those desired by the system.

A top-down approach is suitable only if the specifications of the system are clearly known and the system development is from scratch. Hence, it is a reasonable approach if a waterfall type of process model is being used. However, if a system is to be built from an existing system, a bottom-up approach is more suitable, as it starts from some existing components. So, for example, if an iterative enhancement type of process is being followed, in later iterations, the bottom-up approach could be more suitable (in the first iteration a top-down approach can be used.) Pure top-down or pure bottom-up approaches are often not practical. A common approach to combine the two approaches is to provide a layer of abstraction.

4.2. Module-Level Concepts

A *module* is a logically separable part of a program. It is a program unit that is discreet and identifiable with respect to compiling and loading. In terms of common programming language

constructs, a module can be a macro, a function, a procedure (or subroutine. In systems using functional abstraction, a module is usually a procedure of function or a collection of these. To produce modular designs, some criteria must be used to select modules so that the modules support well-defined abstractions and are solvable and modifiable separately. In a system using functional abstraction, coupling and cohesion are two modularization criteria.

4.2.1 Coupling

Two modules are considered independent if one can function completely without the presence of other. If two modules are independent, they are solvable and modifiable separately. However, all the modules in a system cannot be independent of each other, as they must interact so that together they produce the desired behavior of the system. The more connections between modules more knowledge about one module is required to understand or solve the other module. Hence, the fewer and simpler the connections between modules, the easier it is to understand one without understanding the other.

Coupling between modules is the strength of interconnections between modules or a measure of interdependence among modules. In general, the more we must know about module A in order to understand module B, the more closely connected A is to B; "Highly coupled" modules are joined by strong interconnections, while "loosely coupled" modules have weak interconnections. Independent modules have no interconnections. To solve and modify a module separately, we would like the module to be loosely coupled with other modules. The choice of modules decides the coupling between modules. Coupling is an abstract concept and is not easily quantifiable. So, no formulas can be given to determine the coupling between two modules. However, some major factors can be identified as influencing coupling between modules. Among them the most important are the type of connection between modules, the complexity of the interface, and the type of information flow between modules.

Coupling increases with the complexity of the interface between modules. To keep coupling low we would like to minimize the number of interfaces per module and the complexity of each interface. An interface of a module is used to pass information to and from other modules. Coupling would increase if a module is used by other modules via an indirect and obscure

interface, like directly using the internals of a module or using shared variables. Complexity of the interface is another factor affecting coupling. The more complex each interface is, the higher will be the degree of coupling. For example, complexity of the entry interface of a procedure depends on the number of items being passed as parameters. For example, if a field of a record is needed by a procedure, often the entire record is passed, rather than just passing that field of the record. By passing the record we are increasing the coupling unnecessarily. Essentially, we should keep the interface of module as simple and small as possible.

The type of information flow along the interfaces is the third major factor affecting coupling. There are two kinds of information that can flow along an interface: data or control, Passing or receiving control information means that the action of the module will depend on this control information, which makes it more difficult to understand the module and provide its abstraction. Transfer of data information means that a module passes as input some data to another module and gets in return some data as output. This allows a module to be treated as a simple input output function that performs some transformation on the input data to produce the output data. In general, interfaces with only data communication result in the lowest degree of coupling, followed by interfaces that only transfer control data. Coupling is considered highest if the data is hybrid, that is, some data items and some control items are passed between modules.

4.2.2 Cohesion

With cohesion, we are interested in determining how closely the elements of a module are related to each other. Cohesion of a module represents how tightly bound the internal elements of the module are to one another. Cohesion and coupling are clearly related. Usually, the greater the cohesion of each module in the system, the lower the coupling between modules is. There are several levels of cohesion:

- Coincidental	- Logical	- Temporal	-Procedural
- Communicational	- Sequential	- Functional.	

Coincidental is the lowest level, and functional is the highest. Functional binding is much stronger than the rest, while the first two are considered much weaker than others. Coincidental cohesion occurs when there is *no* meaningful relationship among the elements of a module. Coincidental cohesion can occur if an existing program is modularized by chopping it into pieces and making different pieces modules. If *a mo*dule is created *to* save duplicate *code* by combining some part of-code that Interface occurs at many different places, that module is likely to have coincidental cohesion. In this situation, the statements in the module have no relationship with each other, and if one of the modules using the code needs to be modified and this modification includes the common code, it is likely that other modules using the code do not want the code modified. Consequently, the modification of this "common module" may cause other modules to behave incorrectly. It is poor practice to create a module merely to avoid duplicate code.

A module has **logical cohesion** if there is some logical relationship between the elements of a module, and the elements perform functions that fall in the same logical class. A typical example of this kind of cohesion is a module that performs all the inputs or all the outputs. In such a situation, if we want to input or output a particular record, we have to somehow convey this to the module. Often, this will be done by passing some kind of special status flag, which will be used to determine that statements to execute in the module. This results in hybrid information flow between modules, which is generally the worst form of coupling between modules. Logically cohesive modules should be avoided, if possible.

Temporal cohesion is the same as logical cohesion, except that the elements are also related in time and are executed together. Modules that perform activities like "initialization," "clean-up," and "termination" are usually temporally bound. Temporal cohesion is higher than logical cohesion, because the elements are all executed together. This avoids the problem of passing the flag, and the code is usually simpler.

A **procedurally cohesive** module contains elements that belong to a common procedural unit. For example, a loop or a sequence of decision statements in a module may be combined to form a separate module.

A module with **communicational cohesion** has elements that are related by a reference to the same input or output data. That is, in a communicational bound module, the elements are together because they operate on the same input or output data. An example of this could be a module to "print and punch record. Communicational cohesive modules may perform more than one function. by a reference to the same input or output data. An example of this could be a module to "print and punch record."

When the elements are together in a module because the output of one forms the input to another, we get **sequential cohesion**. If we have a sequence of elements in which the output of one forms the input to another, sequential cohesion does not provide any guidelines on how to combine them into modules. A sequentially bound module may contain several functions or parts of different functions. Sequentially cohesive modules bear a close resemblance to the problem structure.

Functional cohesion is the strongest cohesion. In a functionally bound module, all the elements of the module are related to performing a single function. By function, we do not mean simply mathematical functions; modules accomplishing a single goal are also included. Functions like "compute square root" and "sort the array" are clear examples of functionally cohesive modules.

How does one determine the cohesion level of a module? There is no mathematical formula that can be used. We have to use our judgment for this. A useful technique for determining if a module has functional cohesion is to write a sentence that describes fully and accurately, the function or purpose of the module. The following tests can then be made:

1. If the sentence is a compound sentence, if it contains has more than one verb, the module is probably performing more than one function, and it probably has sequential or communicational cohesion.
2. If the sentence contains words relating to time, like "first," "next," "when" and "after", the module probably has sequential or temporal cohesion.
3. If the predicate of the sentence does not contain a single specific object following the verb (such as "edit all data") the module probably has logical cohesion.

4. Words like "initialize," and "cleanup" imply temporal cohesion.
5. Modules with functional cohesion. can always be described by a simple sentence.

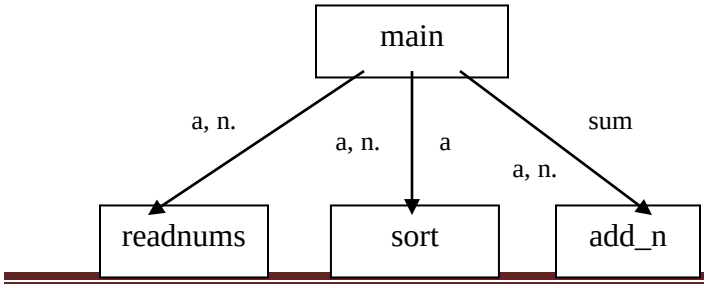
4.3. Design Notation and Specification

While designing, a designer needs to record his thoughts and decisions and to represent the design so that he can view it and play with it. For this, design notations are used. Design notations are largely meant to be used during the process of design and are used to represent design or design decisions. They are meant largely for the designer so that he can quickly represent his decisions in a compact manner that he can evaluate and modify. These notations are frequently graphical. Once the designer is satisfied with the design he has produced, the design is to be precisely specified in the form of a document. To specify the design, specifications are used.

4.3.1. Structure Charts

For a function-oriented design, the design can be represented graphically by structure charts. The structure of a program is made up of the modules of that program together with the interconnections between modules. The structure chart of a program is a graphic representation of its structure. In a structure chart a module is represented by a box with the module name written in the box. An arrow from module A to module B represents that module A invokes module B. B is called the *subordinate* of A, and A is called the *superiordinate* of B. The arrow is labeled by the parameters received by B as input and the parameters returned by B as output, with the direction of flow of the input and output parameters represented by small arrows. The parameters can be shown to be data (unfilled circle at the tail of the label) or control (filled circle at the tail). As an example consider the structure of the following program, whose structure is shown in Figure 4.1.

```
main()
{
int sum, n, N, a[MAX];
readnums(a, &N); sort(a, N); scanf(&n);
sUm = add_n(a, n); printf(sum);
}
readnums(int a[], int *N)
{ - - -
  - - -
}
sort(int a[], int N)
```



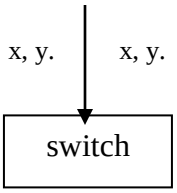


Figure 4.1: The structure of the chart of the sort program

There are some situations where the designer may wish to communicate certain procedural information explicitly, like major loops and decisions. Such information can also be in a structure chart. A loop can be represented by a looping arrow. In Figure given below, module A calls module C and D repeatedly. All the subordinate modules activated within a common loop are enclosed in the same looping arrow.

Major decisions can be represented similarly. For example, if the invocation of modules C and D in module A depends on the outcome of some decision, that is represented by a small diamond in the box for A, with the arrows joining C and D coming out of this diamond, as shown in Figure.

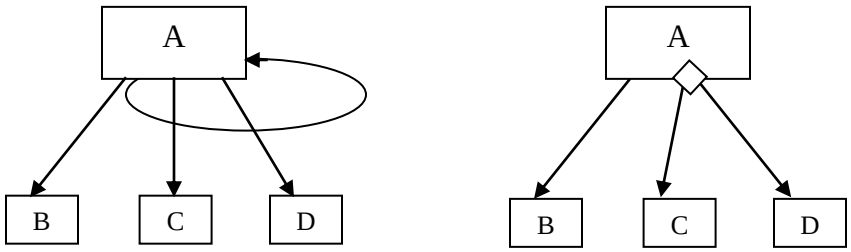


Figure 4.2: Iteration and decision representation

Modules in a system can be categorized into few classes. There are some modules that obtain information from their subordinates and then pass it to their superiordinate. This kind of module is an *input module*. Similarly, there are *output modules* that take information from their superiordinate and pass it on to its subordinates. As the name suggests, the input and output modules are typically used for input and output of data The input modules get the data from the

sources and get it ready to be processed, and the output modules take the output produced and prepare it for proper presentation to the environment.

Then there are modules that exist solely for the sake of transforming data into some other form. Such a module is called a *transform module.* Most of the computational modules typically fall in this category. Finally, there are modules whose primary concern is managing the flow of data to and from different subordinates. Such modules are called *coordinate modules.* The structure chart representation of the different types of modules is shown in Figure 4.3. A module can perform functions of more than one type of module.

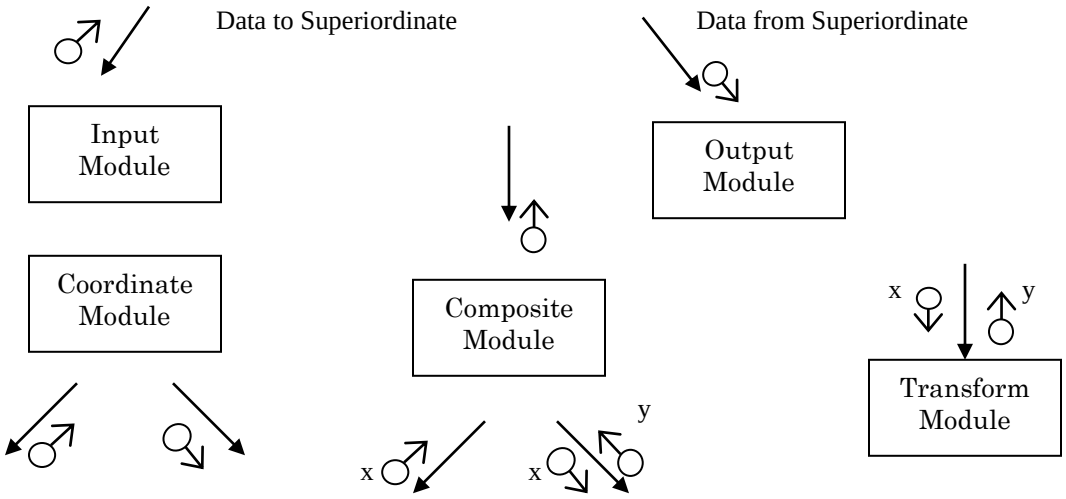


Figure 4.3: Different types of modules

A structure chart is very useful while creating the design. That is a designer can make effective use of structure charts to represent the models creating while he is designing. However, it is not very useful for representing the final design, as it does not give all the information needed about the design. For example, it does not specify the scope, structure of data, specification of each module, etc

4.3.2. Specifications

A design document is the means by which the design is communicated; it should contain all information relevant to future phases. A design specification should contain

1. Problem specification

2. Major data structures
3. Modules and their specifications
4. Design decisions

Module specification is the major part of system design specification. All modules in the system should be identified when the system design is complete, and these modules should be specified in the document. During system design only the module specification is obtained, because the internal details of the modules arc defined later. To specify a module, the design document must specify (a) the *interface of the module* (all data items, their types, and whether they are for input and/or output), (b) the *abstract behavior* of the module *(what* the module does) by specifying the module's functionality or its input/output behavior, and (c) all other modules used by the module being specified-this information is quite useful in maintaining and understanding the design.

4.4. Structured Design Methodology

Structured Design Methodology (SDM) views every software system as having some inputs that are converted into the desired outputs by the software system. The software is viewed as a transformation function that transforms the given inputs into the desired outputs, and the central problem of designing software systems is considered to be properly designing this transformation function. Due to this view of software, the structured design methodology is primarily function-oriented and relies heavily on functional abstraction and functional decomposition.

In properly designed systems, it is often the case that a module with subordinate does not actually perform much computation. The bulk of actual computation is performed by its subordinates, and the module itself largely coordinates the data flow between the subordinates to get the computation done. The subordinates in turn can get the bulk of their work done by their subordinates until the "atomic" modules, which have no subordinates, are reached. *Factoring* is the process of decomposing a module so that the bulk of its work is done by its subordinates. There are four major steps in this strategy:

1. Restate the problem as a data flow diagram
2. Identify the input and output data elements
3. First-level factoring

4. Factoring of input, output, and transform branches

4.4.1. Restate the Problem as a Data Flow Diagram

To use the SDM, the first step is to construct the data flow diagram for the problem there is a fundamental difference between the DFDs drawn during requirements analysis and during structured design. In the requirements analysis, a DFD is drawn to model the problem domain. The analyst has little control over the problem, and hence his task is to extract from the problem all the information and then represent it as a DFD. During design, the designer is dealing with the solution domain; the designer has complete freedom in creating a DFD that will solve the problem stated in the SRS.

The general rules of drawing a DFD remain the same As an example, consider the problem of determining the number of different words in an input file. The data flow diagram for this problem is shown n Figure 4.4 This problem as only one input data stream, the input file, while the desired output is the count of different words n the file. o transform the input to the desired output, the first thing we do is form a list of all the words in the file. It is best to then sort the list, as this will make identifying different words easier. This sorted list is then used to count the number of different words and the output of this transform is the desired count, which is then printed. This sequence of data transformation is what we have in the data flow diagram.

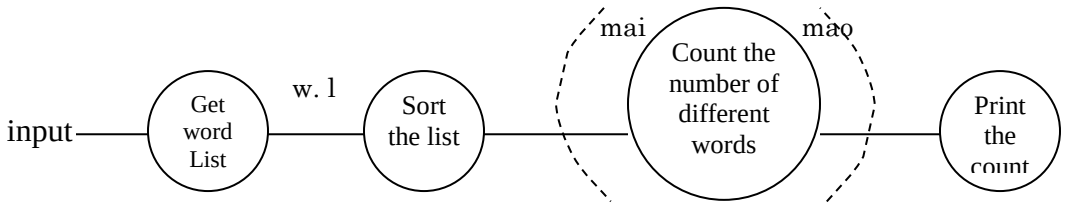


Figure 4.4: DFD for the word-counting problem

4.4.2. Identify the Most Abstract Input and Output Data Elements

Most systems have some basic transformations that perform the required operations. However, in most cases the transformation cannot be easily applied to the actual physical input and produce the desired physical output. Instead, the input is first converted into a form on which the transformation can be applied with ease. Similarly, the main transformation modules often produce outputs that have to be converted into the desired physical output. The goal of this second step is to separate the transforms in the data flow diagram. For this separation, once the data flow diagram is ready, the next step is to identify the highest abstract level of input and output.

The most abstract input data elements(MAI) are those data elements in the data flow diagram that are furthest removed from the physical inputs but it can still be considered inputs to the system. The most abstract input data elements often have little resemblance to the actual physical data. These are often the data elements obtained after operations like error checking, data validation, proper formatting, and conversion are complete.

Similarly, we identify the *most abstract output data elements* (MAO) by starting from the outputs in the data flow diagram and traveling toward the inputs. These are the data elements that are most removed from the actual outputs but can still be considered outgoing. The MAO data elements may also be considered the logical output data items.

There will usually be some transforms left between the most abstract input and output data items. These *central transforms* perform the basic transformation for the system, taking the most abstract input and transforming it into the most abstract output.

Consider the data flow diagram shown in Figure 5.5. The arcs in the data flow diagram are the most abstract input and most abstract output. The choice of the most abstract input is obvious. We start following the input. First, the input file is converted into a word list, which is essentially the input in a different form. The sorted word list is still basically the input, as it is still the same list, in a different order. This appears to be the most abstract input because the next data (i.e., count) is not just another form of the input data. The choice of the most abstract output is even more obvious; count is the natural choice (a data that is a form of input will not usually be a

candidate for the most abstract output). Thus we have one central transform, count-the-number-of-different-words, which has one input and one output data item.

4.4.3. First-Level Factoring

Having identified the central transforms and the most abstract input and output data items, we are ready to identify some modules for the system. We first specify a main module, whose purpose is to invoke the subordinates. The main module is therefore a coordinate module. For each of the most abstract input data items, an immediate subordinate module to the main module is specified. Each of these modules is an input module, whose purpose is to deliver to the main module the most abstract data item for which it is created.

Similarly, for each most abstract output data item, a subordinate module that is an output module that accepts data from the main module is specified. Each of the arrows connecting these input and output subordinate modules are labeled with the respective abstract data item flowing in the proper direction. Finally, for each central transform, a module subordinate to the main one is specified. These modules will be transform modules, whose purpose is to accept data from the main module, and then return the appropriate data back to the main module. The data items coming to a transform module from the main module are on the incoming arcs of the corresponding transform in the data flow diagram. The data items returned are on the outgoing arcs of that transform. Note that here a module is created for a transform, while input/output modules' are created for data items. The structure after the first-level factoring of the word-counting problem (its data flow diagram was given earlier) is shown in Figure 4.5.

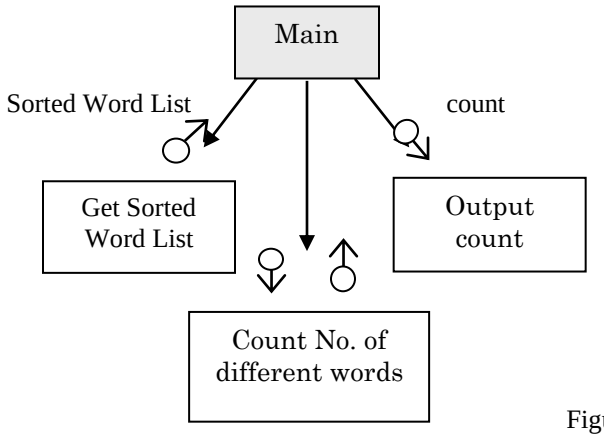


Figure 4.5: First-level factoring

In this example, there is one input module, which returns the sorted word list to the main module. The output module takes from the main module the value of the count. There is only one central transform in this example, and a module is drawn for that. Note that the data items traveling to and from this transformation module are the same as the data items going in and out of the central transform. The main module is the overall control module, which win form the main program or procedure in the implementation of the design. It is a coordinate module that invokes the input modules to get the most abstract data items, passes these to the appropriate transform modules, and delivers the results of the transform modules to other transform modules until the most abstract data items are obtained. These are then passed to the output modules.

4.4.4. Factoring the Input, Output, and Transform Branches

The first-level factoring results in a very high-level structure, where each subordinate module has a lot of processing to do. To simplify these modules, they must be factored into subordinate modules that will distribute the work of a module. Each of the input, output, and transformation modules must be considered for factoring.

The purpose of an input module, as viewed by the main program, is to produce some data. To factor an input module, in the data flow diagram that produced the data item is now treated as a central transform. The process performed for the first-level factoring is repeated here with this new central transform, with the input module being considered the main module. A subordinate input module is created for each input data stream coming into this new central transform, and a subordinate transform module is created for the new central transform. The new input modules now created can then be factored again, until the physical inputs are reached. Factoring of input modules will usually not yield any output subordinate modules.

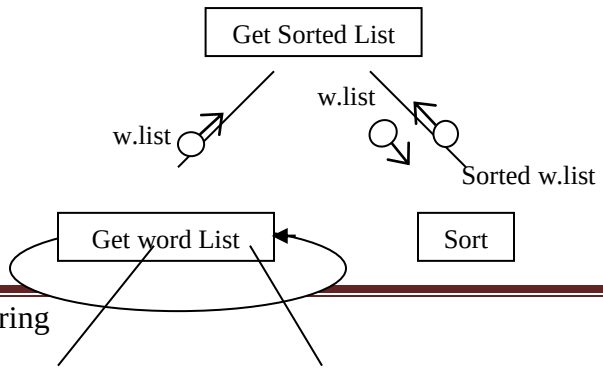
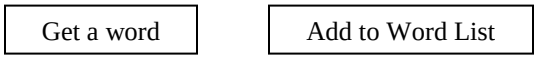


Figure 4.6: Factoring the input module



The factoring of the input module get-sorted-list in the first-level structure is shown in Figure 4.6. The transform producing the input returned by this module (i.e., the sort transform) is treated as a central transform. Its input is the word list. Thus, in the first factoring we have an input module to get the list and a transform module to sort the list. The input module can be factored further, as the module needs to perform two functions, getting a word and then adding it to the list. Note that the looping arrow is used to show the iteration. The factoring of the output modules is symmetrical to the factoring of the input modules. For an output module we look at the next transform to be applied to the output to bring it closer to the ultimate desired output. This now becomes the central transform, and an output module is created for each data stream. Factoring the central transform is essentially an exercise in functional decomposition and will depend on the designers' experience and judgment. One way to factor a transform module is to treat it as a problem in its own right and start with a data flow diagram for it. The factoring of the central transform count-the-number-of-different-words is shown in Figure 4.7.

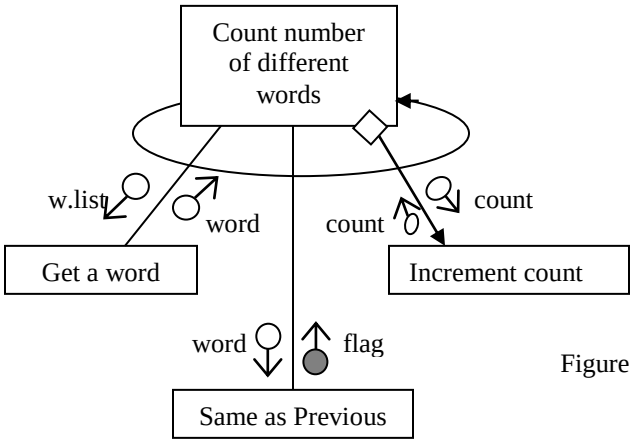


Figure 4.7: Factoring the central transform module

This was a relatively simple transform, and we did not need to draw the data flow diagram. To determine the number of words, we have to get a word repeatedly, determine if it is the same as

the previous word (for a sorted list, this checking is sufficient to determine if the word is different from other words), and then count the word if it is different. For each. of the three different functions, we have a subordinate module, and we get the structure shown in Figure 4.7.

4.4.5. Design Heuristics.

The design steps mentioned earlier do not reduce the design process to a series of steps that can be followed blindly. The strategy requires the designer to exercise sound judgment and common sense. The basic objective is to make the program structure reflect the problem as closely as possible. Here we mention some heuristics that can be used to modify the structure, if necessary.

Module size is often considered the indication of module complexity. In terms of the structure of the system, modules that are very large may not be implementing a single function and can therefore be broken into many modules, each implementing a different function. On the other hand, modules that are too small may not require any additional identity and can be combined with other modules.

However, the decision to split a module or combine different modules should not be based on size alone. Cohesion and coupling of modules should be the primary guiding factors. A module should be split into separate modules only if the cohesion of the original module was low, the resulting modules have a higher degree of cohesion, and the coupling between modules doesn't *increase. Similarly,* two or more modules should be combined only if the resulting module has *a* high degree of cohesion *and* the coupling of the resulting module is not greater than the coupling of the sub-modules. In general, if the module should contain LOC between 5 and 100. Above 100 and less than 5 LOC is not desirable.

Another factor to be considered is "**fan-in" and "fan-out**" of modules. Fan-in of a module is the number of arrows coming towards the module indicating the number of superiordinates. Fan-out of a module is the number of arrows going out of that module; indicating the number of subordinates for that module. A very-high fan-out is not desirable as it means that the module has to control and co-ordinate too many modules. Whenever possible, fan-in should be maximized. In general, the fan-out should not be more than 6.

Another important factor that should be considered is the **correlation of the scope of effect and scope of control.** The scope of effect of a decision (in a module) is collection of all the modules that contain any processing that is conditional that decision or whose invocation is dependent on the outcome of the decision; The scope of control of a module is the module itself and all *its* subordinates (just the immediate subordinates). The system is usually simpler when the scope of effect of a decision is a subset of the scope of control of the module in which decision is located.

4.5. Design Validation/Verification

The output of the system design phase should be verified before proceeding with the activities of the next phase. Unless the design is specified in a formal, executable language, it can not be executed for verification. The most common approach for verification is design reviews.

4.5.1. Design Reviews

The purpose of design reviews is to ensure that the design satisfies the requirements and is of good quality. If errors are made in design phase, they will ultimately reflect themselves in the code and the final system. It is best if the design errors are detected early before they manifest themselves in the final system in order to reduce the cost of fixing bugs. Detecting the errors is the aim of the design review.

In design review, a group of people get together to discuss the design with the aim of revealing design errors or undesirable properties. The review group should include a member of both the system design team and the detail design team, the author of the requirement document and an independent software quality engineer. Each member studies design before the meeting and with the aid of a checklist marks the items that the reviewer feels are incorrect or need clarification. The member asks questions and the chief designer tries to explain the situation. During the course of discussion, design errors if any are revealed.

It should be kept in mind that the aim of the meeting is to uncover design errors and not to try to fix them. Fixing is done later. Also, the psychological frame of mind should be healthy and the designer should not be put in a defensive position. The meeting ends with the list of action items which are later acted on by the design team.

4.5.2. Automated Cross-Checking

If the design is expressed in a language designed for machine processing, most consistency checking can be automated. For example, if a language like PDL is used, the design can be compiled to check for consistency.

PART A (Multiple Choice Questions)
Application

1. For a function-oriented design, the design can be represented graphically by ____
A. Structure Charts
B. System Chart
C. DFD
D. Design Chart

2. An _____of a component describes the external behavior of that component without bothering with the internal details that produce the behavior.
A. Coupling
B. Cohesion
C. Abstraction
D. Factoring

3. In a system using functional abstraction, which are the two modularization criteria.
A. Coupling and cohesion
B. Co-ordinate and transform
C. Top down and bottom up
D. Most abstract inputs and Most abstract output

4. *What defines* the strength of interconnections between modules or a measure of interdependence among modules.
A. Coupling
B. Cohesion
C. Abstraction
D. Factoring

5. Which represents how tightly the internal elements of the module are bound to one another.
A. Coupling
B. Cohesion
C. Abstraction
D. Factoring

6. _____ is the lowest level of cohesion

A. **Coincidental**
B. Logical
C. Temporal
D. Procedural

7. _____ is the highest level of cohesion
A. Procedural
B. Communicational
C. Sequential
D. Functional

8. In a module if the elements are related in time and are executed together,then it is bound to which cohesion
A. Coincidental
B. Logical
C. **Temporal**
D. Procedural
9. Which cohesion occurs when there is *no* meaningful relationship among the elements of a module.
A. **Coincidental**
B. Logical
C. Temporal
D. Procedural
10. Which cohesion occurs , When the elements are together in a module because the output of one forms the input to another
A. Procedural
B. Communicational
C. **Sequential**
D. Functional
11. Which of the following is not a type of module
A. Co-ordinate
B. Output
C. Logical
D. transform

12. Words like "initialize," and "cleanup" imply _____cohesion.
A. Coincidental
B. Logical
C. **Temporal**
D. Procedural

13. modules that obtain information from their subordinates and then pass it to their superiordinate

A. Co-ordinate
B. Transform
C. Input
D. Output

14. *modules* that take information from their superiordinate and pass it on to its subordinates.
A. Co-ordinate
B. Transform
C. Input
D. Output
15. modules whose primary concern is managing the flow of data to and from different subordinates
A. Co-ordinate
B. Transform
C. composite
D. Output

16. The main module is a _____module.
A. composite
B. Co-ordinate
C. Transform
D. Output

17. Which is the major part of system design specification.
A. Module specification
B. First level factoring
C. Coupling
D. Cohesion

18. There are _____ major steps in Structured Design Methodology (SDM)
A. Seven
B. Six
C. Five
D. **Four**
19. First-level factoring and Factoring of input, output, and transform branches are used in ___
A. Structured Design Methodology (SDM)
B. Design heuristics
C. Functional abstraction
D. Design Validation

20. ___is the process of decomposing a module so that the bulk of its work is done by its subordinates.

A. Factoring
B. *abstraction*
C. modularization
D. coupling

21. _____is considered the indication of module complexity.

A. Module size
B. fan-in
C. fan-out
D. factoring

22. ______of a module is the number of arrows coming towards the module indicating the number of superiordinates.

A. fan-in
B. fan-out
C. cohesion
D. co-relation

23. _____of a module is the number of arrows going out of that module; indicating the number of subordinates for that module.

A. fan-in
B. fan-out
C. cohesion
D. co-relation

24. In general, the fan-out should not be more than _____.

A. 4
B. 8
C. **6**
D. 10

25. What is the aim of the design review.

A. Detecting the errors
B. Modifying the design
C. Restating the problem
D. Producing a Structured chart

Part-B
Understanding

1. Illustrate the meaning of abstraction? Explain two common abstraction mechanism for

software system
2. Explain the two modularizing criteria for functional abstraction
3. Identify the meaning of cohesion. Describe the different levels of cohesion
4. Explain any two types of cohesion.
5. Describe the lowest and strongest level of cohesion with suitable example
6. Explain different types of modules in a structure chart.
7. Interpret the meaning of coupling? Explain the factors that affect coupling.

Application

8. Discover the input, output and transform branch factoring with example
9. Illustrate the various factors considered in design heuristics of detailed design
10. Develop the differences between fan in and fan out factoring design heuristic
11. Write a note on SDM strategy.
12. Demonstrate the concept of Factoring? Explain First level factoring with example
13. Write a note on verification in the detailed design phase.

UNIT-4
CHAPTER-5
DETAILED DESIGN

In system design we concentrate on the modules in a system and how they interact with each other. The specifications of a module are often communicated by its name, the English phrase with which we label the module. In previous examples, we have used words like "sort" and "assign" to communicate the functionality of the modules. In a design document, a more detailed specification is given by explaining in natural language what a module is supposed to do. These non-formal methods of specification can lead to problems during coding, because, the coder is a different person from the designer. Even if the designer and the coder are the same person, problems can occur, as the design can take a long time, and the designer may not remember precisely what the module is supposed to do.

The first step before the detailed design or code for a module can be developed is that the specification of the module be given precisely. Once the module is specified, the internal logic for the module that will implement the given specifications can be decided.

5.1 Module Specifications

The specifications of a module should be *complete.* That is, the given specifications should specify the entire behavior of the module that only correct implementations satisfy the specifications. A related property is that the specifications should be *unambiguous Formal* specifications usually are unambiguous while specifications written in natural languages are likely to be ambiguous.

The specifications should be easily *understandable* and the specification language should be such that specifications can be easily written. This is required for practical reasons and is a much desired property if the specification method is to be used in actual software development.

An important property of specifications is that they should be *implementation independent.* Specifications should be given in an abstract manner independent of the eventual implementation

of the module and should not specify or suggest any particular method for implementation of the module. This property specifically rules out algorithmic methods for specification. The specification should only give the external behavior; the internal details of the module should be decided later by the programmer.

5.1.1 Specifying Functional Modules

The most abstract view of a functional module is to treat it as a black box that takes in some inputs and produces some outputs such that the outputs have a specified relationship with the inputs. Most modules are designed to operate only on inputs that satisfy some constraints. For example, a function that finds the square root of a number may be designed to operate only on the real numbers. In addition, it may require that inputs are positive real numbers.

If the inputs satisfy the desired constraints, the goal of a module is to produce outputs that satisfy some constraints that are often related to the inputs. Hence, to specify the external behavior of a module supporting functional abstraction, one needs to specify the inputs on which the module operates, the outputs produced by the module, and the relationship of the outputs to the inputs.

One method for specifying modules is based on pre and post-conditions. In this method constraints on the input of a module were specified by a logical, assertion on the input state called *pre-condition.* The output was specified as a logical assertion on the output state called *post-condition.* As an example, consider a module sort to be written to sort a list L of integers in ascending order. The pre- and post-condition of this module are:

Pre-condition: non-null L
Post-condition: for all i, 1<= i <= size(L), L[i] <= L[i + 1]

The specification states that if the input state for the module sort is non-null L, the output state should be such that the elements of L are in increasing order. These specifications are not complete. They only state that the final state of the list L (which is the output of the module sort) should be such that the elements are in ascending order. It does not state anything about the implicit requirement of the sort module that the final state of the list L should contain the same

elements as the initial list. In fact, this specification can be satisfied by a module that takes the first element of the list L and copies it on all the other elements. A variation of this approach is to specify assertions for the input and output states, but the assertions for the output can be stated as a relation between the final state and the initial state. In such methods, while specifying the condition on the output, the final state of an entity E is referred to as *Eprime* (the initial state is referred by E itself). Using this notation, a possible specification of the module sort is

sort (L: list of integers)
input: non-null L
output: for all i, 1<= i <= size(L),
L'[i] ~ L'[i+l] and
L' = permutation (L)

This specification, besides the ordering requirement, states that elements of the final list are a permutation of elements of the initial list. This specification will be complete if the module is to be designed only to operate on non-null lists. Often the modules check if the input satisfies the desired constraints. If the constraints are not satisfied, it is treated as an *exception condition,* and some special code is executed. If we want the module to handle exceptions, we need to specify the *exceptional behavior* of the modules.

5.2. Detailed Design

Most design techniques identify the major modules and the major data flow among them. Process Design Language (PDL) is one way in which design can be communicated precisely and completely. PDL is particularly useful when using top-down refinement techniques to design a system or a module.

5.2.1. PDL

PDL has an overall outer syntax of a structured programming language and has a vocabulary of a natural language (English in our case). It can be thought of as "structured English". Because the structure of a design expressed in PDL is formal, using the formal language constructs, some amount of automated processing can be done on such designs. As an example, consider the

problem of finding the minimum and maximum of a set of numbers in a file and outputting these numbers in PDL as shown in Figure given below.

```
minmax (infile)
        ARRAY a
        DO UNTIL end of input
                READ an item to a
        ENDDO
        max, min := first item of a
        DO FOR each item in a
                IF max < item THEN set max to item
                IF min > item THEN set min to item
        ENDDO
END
```

PDL description of the minmax program.

Notice that in the PDL program we have the entire logic of the procedure, but little about the details of implementation in a particular language. To implement this in a language, each of the PDL statements will have to be converted into programming language statements. With PDL, a design can be expressed in whatever level of detail that is suitable for the problem. One way to use PDL is to first generate a rough outline of the entire solution at a given level of detail. When the design is agreed on at this level, more detail can be added. This allows a successive refinement approach, and can save considerable cost by detecting the design errors early during the design phase. It also aids design verification by phases, which helps in developing error-free designs. The structured outer syntax of PDL also encourages the use of structured language constructs while implementing the design. The basic constructs of PDL are similar to those of a structured language.

PDL provides IF construct which is similar to the if-then-else construct of Pascal. Conditions and the statements to be executed need not be stated in a formal language. For a general selection, there is a CASE statement. Some examples of The DO construct is used to indicate repetition. The construct is indicated by:

DO iteration-criteria

one or more statements

ENDDO

The iteration criteria can be chosen to suit the problem, and unlike a formal programming language, they need not be formally stated. Examples of valid uses are:

DO WHILE there are characters in input file

DO UNTIL the end of file is reached

A variety of data structures can be defined and used in PDL such as lists, tables, scalar, and integers. Variations of PDL, along with some automated support, are used extensively for communicating designs.

5.2.2 Logic/Algorithm Design

The basic goal in detailed design is to specify the logic for the different modules that have been specified during system design. Specifying the logic will require developing an algorithm that will implement the given specifications. Here we consider some principles for designing algorithms or logic that will implement the given specifications. An *algorithm* is a sequence of steps that need to be performed to solve a given problem. The problem need not be a programming problem. We can, for example, design algorithms for such activities as cooking dishes (the recipes are nothing but algorithms) and building a table. A *procedure* is a finite sequence of well-defined steps or operations, each of which requires a finite amount of memory and time to complete.

There are a number of steps that one has to perform while developing an algorithm. The starting step in the design of algorithms is *statement of the problem.* The problem for which an algorithm is being devised has to be precisely and clearly stated and .properly understood by the person responsible for designing the algorithm. For detailed design, the problem statement comes from the system design. The next step is development of a mathematical *model* for the problem. In modeling, one has to select the mathematical structures that are best suited for the problem. The next step is the *design of the algorithm.* During this step the data structure and program structure are decided. Once the algorithm is designed, correctness should be verified. No clear procedure can be given for designing algorithms.

The most common method for designing algorithms or the logic for a module is to use the *stepwise refinement technique*. The stepwise refinement technique breaks the logic design problem into a series of steps, so that the development can be done gradually. The process starts by converting the specifications of the module into an abstract description of an algorithm containing a few abstract statements. In each step, one or several statements in the algorithm developed so far are decomposed into more detailed instructions. The successive refinement terminates when all instructions are sufficiently precise that they can easily be converted into programming language statements. The stepwise refinement technique is a top-down method for developing detailed design.

An Example: Let us consider the problem of counting different words in a text file. Suppose that the COUNT module is specified whose job is to determine the count of different words. During detailed design we have to determine the logic of this module so that the specifications are met. We will use the stepwise refinement method for this. For specification we will use PDL, adapted to C-style syntax. A simple strategy for the first step is shown bellow (Figure(a)). The primitive operations used in this strategy are very high-level and need to be further refined (as shown in figure (b)).

```
int count(
word
        _list wl;
FILE file)
{
        read file into wl;
        sort (wl);
        count = different_words(wl)
        printf (count);
}
```

Figure (a): Strategy for the first step in stepwise refinement

```
read_from_file(FILE file, word_list wl)
{
        initialize wl to empty;
        while not end-of-file{
                get_a_word from file
                add word to wI
        }
}
```

Figure (b): Refinement of the reading operation.

Specifically, there are three operations that need refinement. These are

(1) read file into the word list, whose purpose is to read all the words from the file and create a word list,

(2) sort(wl), which sorts the word list in ascending order, and

(3) count different words from a sorted word list. So far, only one data structure is defined: the word list. As refinement proceeds, more data structures might be needed.

In the next refinement step, we should select one of the three operations to be refined-and further elaborate it. In this step we will refine the reading procedure. One strategy of implementing the read module is to read words and add them to the word list. This is shown in Figure (b). For the next refinement step we select the counting function. A strategy for implementing this function is shown in Figure (c). Similarly, we can refine the sort function. Once these refinements are done, we have a design that is sufficiently detailed and needs no further refinement. For more complex problems many successive refinements might be needed for a single operation.

```
int different_words (word_list wl)
{
        word last, cur;
        int cnt;
        last = first word in wl
        cnt = 1;
        while not end of list {
                cur = next word from wl
                if (cur <> last) {
                        cnt = cnt + 1;
                        last = cur;
                }
        }
        return (cnt)
}
```

Figure (c). Refinement of the function different_words.

5.3Verification

There are a few techniques available to verify that the detailed design is consistent with the system design. The focus of verification in the detailed design phase is on showing that the detailed design meets the specifications laid down in the system design. Validating that the system as designed is consistent with the requirements of the system is not stressed during detailed design. The three verification methods we consider are design walkthroughs, critical design review and consistency checkers.

5.3.1 Design Walkthroughs

A *design walkthrough* is a manual method of verification. A design walkthrough is done in an informal meeting called by the designer or the leader of the designer's group. The walkthrough group is usually small and contains, along with the designer, the group leader and/or another designer of the group. The designer might just get together with a colleague for the walkthrough or the group leader might require the designer to have the walkthrough with him. In a walkthrough the designer explains the logic step by step, and the members of the group ask questions, point out possible errors or seek clarification. A beneficial side effect of walkthroughs is that in the process of articulating and explaining the design in detail, the designer himself can uncover some of the errors. Walkthroughs are essentially a form of peer review. Due to its informal nature, they are usually not as effective as the design review.

5.3.2 Critical Design Review

The purpose of *critical design review* is to ensure that the detailed design satisfies the specifications laid down during system design. It is very desirable to detect and remove design errors early, as the cost of removing them later can be considerably more than the cost of removing them at design time. Detecting errors in detailed design is the aim of critical design review.

The critical design review process is similar to the other reviews, in that a group *of* people get together to discuss the design with the aim of revealing designs errors or undesirable properties.

The review group includes, besides the author of detailed design, a member of the system design team, the programmer responsible for ultimately coding the module(s) under review, and an independent software quality engineer. That is, each member studies the design beforehand and with aid of a checklist marks items that the reviewer feels are incorrect or need c1arification. The members ask questions and the designer tries to explain the situation. During the discussion design errors are revealed. As with any review, it should be kept in mind that the aim of the meeting is to uncover design errors, not try to fix them. Fixing is done later. Also, the psychological frame of mind should be healthy, and the designer should not be put in a defensive position. The meeting should end with a list of action items, to be acted on later by the designer. The use of checklists, as with other reviews, is considered important for the success of the review.

5.3.3. Consistency Checkers

Design reviews and walkthroughs are manual processes; the people involved in the review and walkthrough determine the errors in the design. If the design is specified in PDL or some other formally defined design language, it is possible to detect some design defects by using consistency checkers. *Consistency checkers* are essentially compilers that take as input the design specified in a design language (PDL). Clearly, they cannot produce executable code because the inner syntax of PDL allows natural language and many activities specified in the natural language. However, the module interface specifications (which belong to outer syntax) are specified formally. A consistency checker can ensure that any modules invoked or used by a given module actually exist in the design and that the interface used by the caller is consistent with the interface definition of the called module. It can also check if the used global data items are defined globally in the design.

CHAPTER-6

CODING

The goal of the coding or programming phase is to translate the design of the system produced during the design phase into code in a given programming language, which can be executed by a computer and that performs the computation specified by the design. The coding phase affects both testing and maintenance profoundly. As we saw earlier, the time spent in coding is a small percentage of the total software cost, while testing and maintenance consume the major percentage. Thus, it should be clear that the goal during coding should *not* be to reduce the implementation cost, but the goal should be to reduce the cost of later phases, even if it means that the cost of this phase has to increase. In other words, the goal during this phase is *not* to simplify the job of the programmer. Rather, the goal should be to simplify the job of the tester and the maintainer.

During implementation, it should be kept in mind that, the programs should not be constructed so that they are easy to write, but so that they are easy to read and understand.

6.1. Programming Practice

The primary goal of the coding phase is to translate the given design into source code in a given programming language, so that code is simple, easy to test, and easy to understand and modify. Simplicity and clarity are the properties a programmer should strive for. Good programming is a skill that can only be acquired by practice. However, much can be learned from the experience of others, and some general rules and guidelines can be laid for the programmer. Good programming (producing correct and simple programs) is a practice independent of the target programming language.

6.1.1 Top-Down and Bottom-Up

In a top-down implementation, the implementation starts from the top of the hierarchy and proceeds to the lower levels. First the main module is implemented, then its subordinates are implemented, and their subordinates, and so on. In a bottom-up implementation, the process is the reverse. The development starts with implementing the modules at the bottom of the hierarchy and proceeds through the higher levels until it reaches the top.

Top-down and bottom-up implementation should not be confused with top-down and bottom-up design. Here, the design is being implemented, and if the design is fairly detailed and complete, its implementation can proceed in either the top-down or the bottom-up manner, even if the design was produced in a top-down manner. Which of the two is used mostly affects testing. All large systems must be built by assembling validated pieces together. The case with software systems is the same. Parts of the system have to first be built and tested before putting them together to form the system. Because parts have to be built and tested separately, the issue of top-down versus bottom-up arises.

.

6.1.2 Structured Programming

The basic objective of the coding activity is to produce programs are easy to understand. It has been argued by many that structured programming practice helps develop programs that are easier to understand. Structured programming is often regarded as "goto-less" programming. Although extensive use of gotos is certainly desirable, structured programs *can* be written with the use of gotos.

A program has a static structure as well as a dynamic structure. The static structure is the structure of the text of the program, which is usually just a linear organization of statements of the program, The dynamic structure of the program is the sequences of statements executed during the execution of the program. In other words, both the static structure and the dynamic behavior are sequences of statements; where the sequence representing the static structure of a program is fixed, the sequence of statements it executes can change from execution to execution.

It will be easier to understand the dynamic behavior if the structure in the dynamic behavior resembles the static structure. The closer the correspondence between execution and text structure, the easier the program is to understand, and the more different the structure during execution, the harder it will be to argue about the behavior from the program text. The goal of structured programming is to ensure that the static structure and the dynamic structures are the

same. That is, the objective of structured programming is to write programs so that the sequence of statements executed during the execution of a program is the same as the sequence of statements in the text of that program. As the statements in a program text are linearly organized, the objective of structured programming becomes developing programs whose control flow during execution is linearized and follows the linear organization of the program text. Clearly, no meaningful program can be written as a sequence of simple statements without any branching or repetition. In structured programming, a statement is not a simple assignment statement, it is a structured statement. The key property of a structured statement is that it has a *single-entry and a single-exit*, That is, during execution, the execution of the (structured) statement starts from one defined point and the execution terminates at one defined point. With single-entry and single-exit statements, we can view a program as a sequence of (structured) statements. And if all statements are structured statements, then during execution, the sequence of execution of these statements will be the same as the sequence in the program text. Hence, by using single-entry and single-exit statements, the correspondence between the static and dynamic structures can be obtained. The most commonly used single-entry and single-exit statements are:

Selection: if B then S1 else S2
if B then SI

Iteration: While B do S
Repeat S until B

Sequencing: S1; S2; S3;.

It can be shown that these three basic constructs are sufficient to program any conceivable algorithm. Modem languages have other such constructs that help linearize the control flow of a program, which makes it easier to understand a program. Hence, programs should be written so that, as far as possible, single-entry, single-exit control constructs is used. The basic goal, as we have tried to emphasize, is to make the logic of the program simple to understand. The basic objective of using structured constructs is to linearize the control flow so that the execution behavior is easier to understand. In linearized control flow, if we understand the behavior of each of the basic constructs properly, the behavior of the program can be considered a composition of the behaviors of the different statements. Overall, it can be said that structured programming, in general, leads to programs that are easier to understand than unstructured programs.

6.1.3 Information Hiding

A software solution to a problem always contains data structures that are meant to represent information in the problem domain. That is, when software is developed resolve a problem, the software uses some data structures to capture the information in the problem domain. Any software solution to a problem contains data structures that represent information in the problem domain. In the problem domain, in general, only certain operations are performed on some information. That is, a piece of information in the problem domain is used only in a limited number of ways in the problem domain. For example, a ledger in an accountant's office has some defined uses: debit, credit, check the current balance, etc. An operation where all debits are multiplied together and then divided by the sum of all credits is typically not performed. So, any information in the problem domain typically has a small number of defined operations performed on it.

When the information is represented as data structures, the same principle should be applied, and only some defined operations should be performed on the data structures. This, essentially, is the principle of information hiding. The information captured in the data structures should be hidden from the test of the system, and only the access functions on the data structures that represent the operations performed on the information should be visible. The other modules access the data only with the help of these access functions.

6.1.4 Programming Style

Here we will list some general rules that can be applied for writing good code.

Names: Selecting module and variable names is often not considered important novice programmers. Most variables in a program reflect some entity in the problem domain, and the modules reflect some process. Variable names should be closely related to the entity they represent, and module names should reflect their activity. It is bad practice to choose cryptic names (just to avoid typing: or totally unrelated names. It is also bad practice to use the same name for multiple purposes.

Control Constructs: As discussed earlier, it is desirable that as much as possible single-entry, single-exit constructs be used. It is also desirable to use a few standard control constructs rather than using a wide variety of constructs, just because the; are available in the language.

Gotos: Gotos should be used sparingly and in a disciplined manner. Only when the alternative to using gotos is more complex should the gotos be used. In any case, alternatives must be thought before finally using a goto. If a goto must be used, forward transfers (or a jump to" later statement) is more acceptable than a backward jump. Use of gotos for exit a loop or for invoking error handlers is quite acceptable.

Information Hiding: As discussed earlier, information hiding should be supported where possible. Only the access functions for the data structures should be made visible while hiding the data structure behind these functions.

User-Defined Types: Modern languages allow users to define data types when such facilities are available, they should be exploited where applicable. For example, when working with dates, a type can be defined for the day of the week. In Pascal, this is done as follows: type days = (Man, Tue, Wed, Thur, Fri, Sat, Sun); Variables can then be declared of this type. Using such types makes the program much clearer than defining codes for each day and then working with codes.

Nesting: The different control constructs, particularly the if-then-else, can be nested. If the nesting becomes too deep, the programs become harder to understand. In case of deeply nested if-then-elses, it is often difficult to determine if statement to which a particular else clause is associated. If possible, deep nesting should be avoided.

ifCl then Sl
else if C2 then S2
else if C3 then S3
else if C4 then *S4;*
If the different conditions are disjoint (as they often are), this structure can be converted into the following structure:

ifC1 *then* Sl;

if C2 then S2;

if C3 then S3;

if C4 then S4;

This sequence of statements will produce the same result as the earlier sequence (if the conditions are disjoint), but it is much easier to understand.

Module Size: A programmer should carefully examine any routine with very few statements (say fewer than 5) or with too many statements (say more than 50). Large modules often will not be functionally cohesive, and too-small modules might incur unnecessary overhead. There can be no hard-and-fast rule about module sizes the guiding principle should be cohesion and coupling.

Module Interface: A module with a complex interface should be carefully examined. Such modules might not be functionally cohesive and might be implementing multiple functions. As a rule of thumb, any module whose interface has more than five parameters should be carefully examined and broken into multiple modules with a simpler interface if possible.

Program Layout: How the program is organized and presented can have great effect on the readability of it. Proper indentation, blank spaces, and parentheses should be used to enhance the readability of programs.

Side Effects: When a module is invoked, it sometimes has side effects of modifying the program state beyond the modification of parameters listed in the module interface definition, for example, modifying global variables. Such side effects should be avoided where possible, and if a module has side effects, they should be properly documented.

Robustness: A program is robust if it does something planned even for exceptional conditions. A program might encounter exceptional conditions in such forms as incorrect input, the incorrect value of some variable, and overflow. A program should try to handle such situations. In general, a program should check for validity of inputs, where possible, and should check for possible

overflow of the data 5ll1lctures. If such situations do arise, the program should not just "crash" or "core jump"; it should produce some meaningful message and exit gracefully.

6.1.5 Internal Documentation

In the coding phase, the output document is the code itself. However, some amount of internal documentation in the code can be extremely useful in enhancing the understandability of programs. Internal documentation of programs is done by the use of comments. All languages provide a means for writing comments in programs. Comments are textual statements that are meant for the program reader and are not executed. Comments, if properly written and kept consistent with the code, can be invaluable during maintenance. The purpose of comments is not to explain in English the logic of the program. The program itself is the best documentation for the details of the logic. The comments should explain what the code is doing, not how it is doing it. Comments should be provided for blocks of code, particularly those parts of code that are hard to follow. Providing comments for modules is most useful, as modules form the unit of testing, compiling, verification and modification. It contains the following information.

1. Module functionality, or what the module is doing.
2. Parameters and their purpose.
3. Assumptions about the inputs, if any.
4. Global variables accessed and/or modified in the module.

An explanation of parameters (whether they are input only, output only, or both input and output; why they are needed by the module; how the parameters are modified) can be quite useful during maintenance. Stating how the global data is affected and the side effects of a module is also very useful during maintenance. In addition other information can be included, depending on the local coding standards. Examples are the name of the author, the date of compilation, and the last date-of modification. It should be pointed out that the prologues are used only if they are kept consistent with the logic of the module. If the module is modified, then the prologue should also be modified, if necessary. A prologue that is inconsistent with the internal logic of the module is probably worse than no prologue at all.

6.2 Verification

Verification of the output of the coding phase is primarily intended for detecting errors introduced during this phase. That is, the goal of verification of the code produced is to show that the code is consistent with the design it is supposed to implement. It should be pointed out that by verification we do not mean proving correctness of programs. Program verification methods fall into two categories—static and dynamic methods. In dynamic methods the program is executed on some test data and the outputs of the program are examined to determine if there are any errors present. Static techniques, on the other hand, do not involve actual program execution on actual numeric data, though it may involve some form of conceptual execution. In static techniques, the program is not compiled and then executed, as in testing. Common forms of static techniques are program verification, code reading, code reviews and walkthroughs, and symbolic execution. In static techniques often the errors are detected directly, unlike dynamic techniques where only the presence of an error is detected.

6.2.1 Code Reading

Code reading involves careful reading of the code by the programmer to detect any discrepancies between the design specifications and the actual implementation. It involves determining the abstraction of a module and then comparing it with its specifications. The process of code reading is best done by reading the code inside-out, starting with the innermost structure of the module.

6.2.2 Static Analysis

Analysis of programs by methodically analyzing the program text is called Static analysis is usually performed mechanically by the aid of software tools. During static analysis the program itself is not executed, but the program text *is* the input to the tools. The aim of the static analysis tools *is* to detect errors or potential errors or to generate information about the structure of the program that can be useful for documentation or understanding of the program. An advantage is that static analysis sometimes detects the errors themselves, not just the presence of errors, as in testing. This saves the effort of tracing the error from the data that reveals the presence of errors. Furthermore, static analysis can provide "warnings" against potential errors and can provide insight into the structure of the program. It is also useful for determining violations of local

programming standards, which the standard compilers will be unable to detect. Extensive static analysis can considerably reduce the effort later needed during testing.

Data flow anomalies are "suspicious" use of data in a program. In general, data flow anomalies are technically not errors, and they may go undetected by the compiler. However, they are often a symptom of an error, caused due to carelessness in typing or error in coding. At the very least, presence of data flow anomalies implies poor coding. Hence, if a program has data flow anomalies, they should be properly addressed.

```
x = a;
x does not appear in any right hand side
x = b;
```

FIGURE 8.2. A code segment.

An example of the data flow anomaly is the **live variable problem**, in which a variable is assigned some value but then the variable is not used in any later computation. Such an assignment to the variable is clearly redundant. Another simple example of this is having two assignments to a variable without using the value of the variable between the two assignments. In this case the first assignment is redundant. For example, consider the simple case of the code segment shown in Figure 8.2. Clearly, the first assignment statement is useless. Perhaps the programmer meant to say y := b in the second statement, and mistyped y as x. In that case, detecting this anomaly and directing the programmer's attention to it can save considerable effort in testing and debugging. In addition to revealing anomalies, data flow analysis can provide valuable information for documentation of programs. For example, data flow analysis can provide information about which variables are modified on invoking a procedure in the caller program and the value of the variables used in the called procedure (this can also be used to make sure that the interface of the procedure is minimum, resulting in lower coupling). This information can be useful during maintenance to ensure that there are no undesirable side effects of some modifications to a procedure.

6.2.3 Symbolic Execution

Here the program is "symbolically executed" with symbolic data. Hence the inputs to the program are not numbers but symbols representing the input data, which can take different

values. The execution of the program proceeds like normal execution, except that it deals with values that are not numbers but formulas consisting of the symbolic input values. The outputs are symbolic formulas of input values. These formulas can be checked to see if the program will behave as expected. This approach is called as symbolic execution.

A simple program to compute the product of three positive integers is shown in Figure 8.3. Let us consider that the symbolic inputs to the function are xi, yi, and zi. We start executing this function with these inputs. The aim is to determine the symbolic values of different variables in the program after "executing" each statement, so that eventually we can determine the result of executing this function.

Example:

```
1. function product (x, y, z: integer): integer;
2. var tmpl, tmp2: integer;
3. begin .
4. tmpl := x*y;
5. tmp2 := y*z;
6. product := tmpl *tmp2/y;
7. end
```

FIGURE 8.3. Function to determine product.

After	Values of the variables					
Statement	x	y	z	tmp1	tmp2	product
1.	xi	yi	zi	?	?	?
2.	xi	yi	zi	xi*yi	?	?
3.	xi	yi	zi	xi*yi	yi*zi	?
4.	xi	yi	zi	xi*yi	yi*zi	(xi*yi)* (yi*zi)/yi

The symbolic execution of the function product

Here there is only one path in the function, and this symbolic execution is equivalent to checking for all possible values of x, y, and z. (Note that the implied assumption is that input values are such that the machine will be able to perform the product and no overflow *will occur.)* Essentially, with only one path and an acceptable symbolic result, we can claim that the program is correct.

Path Conditions

In symbolic execution, when dealing with conditional execution, it is not sufficient to look at the state of the variables of the program at different statements, as a statement will only be executed if the inputs satisfy certain conditions in which the execution of the program will follow a path that includes the statement. To capture this concept in symbolic execution, we require a notion of "path condition." Path condition at a statement gives the conditions the inputs must satisfy for an execution to follow the path so that the statement will be executed. Path condition is a Boolean expression over the symbolic inputs that never contain any program variables. It will be represented in a symbolic execution by pc. Each symbolic execution begins with pc initialized to true. For example, symbolic execution of an if statement of the form if C then S1 else S2 will require two cases to be considered, corresponding to the two possible paths; one where C evaluates to true and S1 is executed, and the other where C evaluates to false and S2 is executed. For the first case we set the path condition pc to

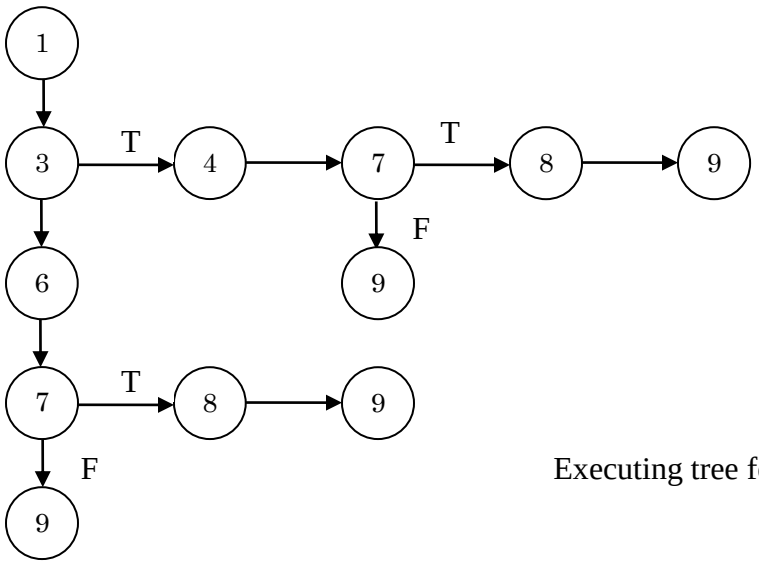
pc ← *pc* ^ C

This is the path condition for the statements in S1. For the second case we set the path condition to *pc* ← *pc*^ ~ C which is the path condition for statements in *S2.*

On encountering the if statement, symbolic execution is said to split into two executions: one following the then part, the other following the else part. Both these paths are independently executed, with their respective path conditions. However, if at any if statement we can show that *pc* implies C or ~C, we do not need to follow both paths, and only the relevant path need be executed. Let us consider an example involving if statements. Figure 8.5 shows a program to determine the maximum of three numbers. The trace of the symbolic execution:

```
1.    function max (x, y, z: integer) : integer
2.    begin
3.        if (x<=y) then
4.            max:=y;
5.        else
6.            max:=x
7.        if max <z then
```

8. max:=z;
9. end;
the code for function max



Executing tree for the function max

The different paths followed during symbolic execution can be represented by an "execution tree." A node in this tree represents the execution of a statement, while an arc represents the transition from one statement to another. For each if statement, there are two arcs from the node corresponding to the if statement, one labeled with T (true) and the other with F (false), for the then and else paths. At each branching, the path condition is also often shown in the tree.

6.2.4Proving Correctness

In proof of correctness, the aim is to prove a program correct. So, correctness is directly established, unlike the other techniques in which correctness is never really established but is implied by the absence of detection of any errors. Proofs are more valuable during program construction, rather than after the program have been constructed. Proving while developing a program may result in more reliable programs that can be proved more easily.

6.2.5 Code Inspections or Reviews

The review process was started with the purpose of detecting errors in the code. Code inspection or reviews are usually held after the successful completion of the coding phase. The entry criteria for code review are that the code must compile successfully and has been passed by other static analysis tools. The documentation to be distributed to the review team members includes

6.2.6 Unit Testing

In this level, different modules are tested against the specification produced during design for the modules. Unit testing is essentially for verification of the code produced during the code phase. That is, the goal of this testing is to test the internal logic of the modules. Due to its close association with coding the coding phase is frequently called ”coding and unit testing” .The goal of unit testing is to test modules or “units”, not the entire software system.

PART A
Multiple Choice Questions
Understanding

1. Functional modules are treated as a _____ that takes in some inputs and produces some outputs

A. White box
B. Block box
C. **Black box**
D. Glass box

2.The method of specifying constraint on input of the module as a logical assertion on the input state is____

A. **Pre-conditon**
B. logical-condition
C. Post-condition
D. E Prime

3.The method which specifies output of a module as a logical assertion on the output state called

A. Pre-condition
B. logical-condition
C. **Post-condition**
D. E Prime

4.Which language is a way to communicate a design precisely and completely.

A. **Process Design Language**
B. Program Design Language
C. Process Definition Language
D. Process Definition Language

5.Which are not consider as verification method for design phase.

A. Design walkthrough
B. 2Critical design review
C. Consistency checkers
D. **Symbolic execution**

6.Which is a manual method of verification in detailed design phase.

A. **Design walkthrough**
B. Critical design review
C. Consistency checkers
D. Path condition

7.____ are compilers which take design specified in PDL as input

A. Consistency Checkers
B. Execution tree
C. Drivers
D. stubs

8.Structured programming is often regarded as_____programming.

A. **Goto-less**
B. Goto with
C. Dynamic
D. stepwise refinement

9.The key property of a structured Programming is that it has a

A. single-entry and a single-exit
C. information hiding
B. goto statement
D. multiple exits

10.Program having static structure implies

A. Linear organization of statements
C. Programs using goto statement
B. Non- Linear organization of statements
D. Sequence of execution of Program ,during execution

11.Expand PDL

A. **Process Design Language**
C. Process Definition Language
B. Program Design Language
D. Process Definition Language

Skill

12. _____is a sequence of steps that need to be performed to solve a given problem.

A. Design
C. Algorithm
B. Path testing
D. internal documentation

13.Detecting the errors in detailed design is the aim of _____

A. Path conditon
C. Algorithm design
B. Critical Design Review
D. Symbolic execution

14.In consistency checker _____ can not be generated.

A. errors
C. output
B. Anomaly
D. **Execution code**

15.what is the linear organization of statement of program referred as.

A. **Static structure**
C. symbolic structure
B. Dynamic structure
D. PDL structure

16.sequence of Statement executed during execution of the program refers to

A. Static structure
C. PDL structure
B. **Dynamic structure**
D. symbolic structure

17.The program is_____, if it does a planned action even for exceptional condition.

A. having Side effect
C. **Robust**
B. static
D. Nested

18.use of comments is for_____

A. Verification
B. Static analysis

C. **Internal documentation**
D. Data flow anomalies

19.____is methodically analysing the program text.
A. code reading
B. **Static analysis**
C. Internal documentation
D. code Inspection

20. Suspicious data in the program is_____
A. comment
B. Path condition
C. Node
D. Data flow anomalies

21. The test associated with coding phase is _____
A. Unit testing
B. System testing
C. Path Testing
D. Symbolic execution test

22. The goal of _____ phase is to translate design to a programming language instruction
A. requirement
B. design
C. Coding
D. testing

23. In which approach code verification, the inputs to the Program is not values /numbers but are symbols and formulas representing input data and Outputs
A. code reading
B. Execution tree
C. Symbolic execution
D. Code inspection

24. *what represents* different paths followed during symbolic execution
A. Node
B. Execution tree
C. Arcs
D. Path condition

25.A_____ in execution tree represents execution of a statement.
A. Symbol
B. Path
C. Node
D. Arcs

PART B

Understanding

1. Explain module specifications in detailed design.
2. Explain PDL with suitable example.
3. Describe the Logic/Algorithm design.
4. Identify the three verification method of a detailed design. Explain any one.
5. Observe and interpret the activities that are undertaken during critical design review.
6. Discuss on the concepts i. Design walkthroughs ii. Consistency checkers.

Application

7. Explain the concept of structured programming.
8. Write a note on data flow anomalies with example
9. Explain any four programming style.
10. Explain internal documentation and what are the information it contains.
11. Explain the symbolic execution and execution tree with an example
12. Explain static analysis and its uses.

UNIT 5
CHAPTER-7

SOFTWARE TESTING AND MAINTENANCE

7.1 INTRODUCTION

In software development process, errors can be injected at any stages during development. We have already discussed about various techniques for detecting and eliminating errors that originate in that phase. However, no method is perfect and it is expected that some errors of the earlier phases will finally reaches to the coding phase. This is because most of the verification methods of earlier phases are manual. Hence the code developed during coding activity is likely to have some requirement errors and design errors, in addition to errors introduced during the coding activity.

During testing, the program to be tested is executed with a set of test cases and the output of the program for the test cases is evaluated to determine if the program is performing as expected. Form this it is clear that testing is used to find out errors rather than to tell the exact nature of the error. Also, the success of the testing process clearly depends upon the test cases used.

Testing is a complex process. In order to make the process simpler, the testing activities are broken into smaller activities. Due to this, for a project, incremental testing is generally performed. In incremental testing process, the system is broken into set of subsystems and these subsystems are tested separately before integrating them to form the system for system testing.

Definitions of testing

- The process of analyzing a software item to detect the differences between existing and required conditions (i.e., bugs) and to evaluate the features of the software items (IEEE 1993)
- The process of analyzing a program with the intent of finding errors.

(Myers 1979)

Some testing principles

- Testing cannot show the absence of defects, only their presence.
- The earlier an error is made, the costlier it is
- The later an error is detected, the costlier it is.

1.2 TESTING FUNDAMENTALS

- **Error:** The term error is used to refer to the discrepancy between computed, observed, or measured value and the true or specified value. In other words, it is the difference between the actual output of software and correct output.
- **Fault:** Fault is a condition that causes a system to fail in performing its required function.

- **Failure:** Failure is the inability of a system to perform a required function according to its specifications. A software failure occurs if the behavior of the software is different from the specified behavior.

7.3 TEST ORACLES

To test any program, we need to have a description of its excepted behavior and a method of determining whether the observed behavior conforms to the excepted behavior. For this we need test oracle.

A test oracle is a mechanism, different from the program itself that can be used to check the correctness of the output of the program for the test cases. Conceptually, we can consider testing a process in which the test cases are given to the test oracles and the program under testing. The output of the two is then compared to determine if the program behaved correctly for the test cases.

Figure 7.1 Illustrates this step.

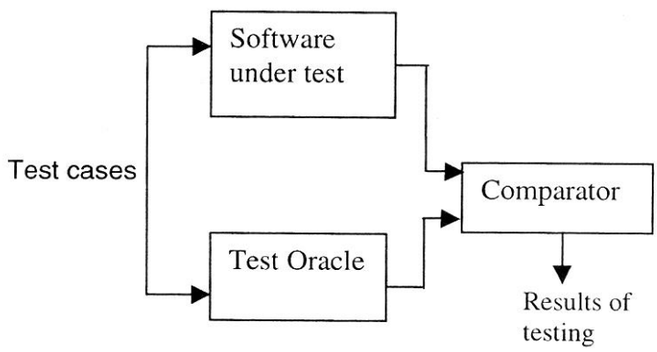


Figure 7.1 Test Oracles

Test oracles generally use the system specifications of the program to decide what the correct behavior of the program should be. To help the oracle to determine the correct behavior, it is important that the behavior of the system be unambiguously specified and the specification itself error free.

1.3 TOP-DOWN AND BOTTOM – UP APPROACHES:

In top-down testing method, we start by testing top of the hierarchy, and we incrementally add modules that it calls and then test the new combined system. This approach requires stubs to be written. A stub is a dummy routine that simulates a module. In Top-Down approach, a module or a collection of modules can not be tested in isolation because they invoke some other modules. To allow the modules to be tested before their subroutines have been coded, stubs simulate the behavior of the subordinates.

The Bottom-up approach starts from the bottom of the hierarchy. First the modules at the lowest level, which have no subordinates, are tested. Then these modules are combined with higher level modules for testing. At any stage of testing all the subordinate modules exist and have been tested earlier. To perform bottom-up testing, drivers are needed to setup the appropriate

environment and invoke the module. It is the job of the driver to invoke the module under testing with the different set of test cases. Both Top-down and Bottom-up approaches are incremental; starting with testing single module and then adding untested modules to those that have been tested, until the entire system is tested.

7.5 TEST CASES

Test cases are required to find out the presence of fault in a system. Test cases are the inputs to the testing process. In order to reveal the correct behavior of the system it is necessary to have a large set of valid test cases. While selecting the test cases the primary objective is to ensure that if there is an error or fault in the program. An ideal test case set is one that succeeds only if there are no errors in the program. One possible ideal set of test cases is one that includes all the possible inputs to the program. This is often called exhaustive testing. However, exhaustive testing is impractical and infeasible, as even for small programs the number of elements in the input domain can be extremely large. Hence, a realistic goal for testing is to select a set of test cases that is close to ideal.

The range and type of test cases to be prepared in order to perform testing depends upon test criterion. A test criterion is the condition that must be satisfied by a set of test cases.

The criterion becomes a basis for test selection. For example: If the criterion is the all statements in the program be executed at least once during testing, then a set of test cases T satisfies this criterion for a program P if the execution of P with T ensure that each statement in P is executed at least once.

7.6 TYPE OF TESTING

There are two basic approaches to testing: functional and structural. In functional testing the structure of the program is not considered. Test cases are decided on the basic of the requirements or specification of the program or module and the internals of the module or the program are not considered for selection of test cases. Functional testing is often called “Black box testing”. In the structural approach, test cases are generated based on the “actual code of the program or the module” to be tested. The structural approach is also known as “Glass box testing”.

7.6.1 Functional Testing

The functional testing procedure is exhaustive testing. One criterion for generating test cases is to generate them randomly. This strategy has little chance of resulting in a set of test cases that is close to optimal. Hence, we need some other criterion for selecting test cases. There are no formal rules for designing test cases for functional testing. However, there are a number of methods or heuristics that can be; used to select test cases. They are

Equivalence class partitioning

In this method the domain of all the inputs are divided into a set of equivalence classes so that if any tests in that class succeed, then every test in that class will succeed. That is, we want to identify classes of test such that the success of one test case in a class implies the success of others. However, without looking at the internal structure of the program, it is impossible to determine such ideal equivalence classes. The equivalence class partitioning method tries to approximate this ideal. Putting inputs for which the behavior pattern of the module is specified to be different into similar group's forms different equivalence classes. For example, the specification module that determines the absolute value for integers specifies one behavior for positive integers and another behavior for negative integers. In this case, we will form two equivalence classes-one consisting of positive integers and the other consisting of negative integers. It is often useful to consider equivalence classes in the output. For an output equivalence class, the goal is to generate test cases such that the output for that test case lies in the output equivalence class. Determining test cases for output classes may be more difficult, but output classes have been found to reveal errors that are not revealed by just considering the input classes.

Boundary value analysis

It has been observed that programs that work correctly for a set of values in an equivalence class fail on some special values. These values often lie on the boundary of the equivalence class. The test values lie on boundaries of equivalence class likely to be "high yield" test cases. Selecting such test cases is the aim of the boundary value analysis. In this analysis, we first choose input for a test case from the equivalence class, such that the input lies on the edge of the equivalence classes. Boundary value test cases are also called "extreme cases".

Hence, we can say that a boundary value test case is a set of input that lies on the edge or boundary of a class of input data or that generates output that lies at the boundary of a class of output data.

Cause-Effect Graphing

The major problem with the equivalence class portioning and boundary value analysis is that they consider each input separately. They do not consider combinations of input. One way to exercise combinations of different input conditions is to consider all valid combinations of equivalence classes of input conditions. Cause-Effect graphing is a technique that aids in selecting combinations of input conditions in a systematic way, such that the number of test cases does not become unmanageably large. This technique starts with identifying causes and effects of the system under testing. A cause is a unique input condition and an effect is a unique output condition. Each condition forms a node in the cause effect graph. The condition should be defined in such a way that they can use to either true or false. For example, an input condition can be "file is empty," which can be set to true by having empty input file, and false by a nonempty file.

After identifying the causes and effects, for each effect we identify the causes that can produce that effect and how the conditions have to be combined using the Boolean operators “and” “or”, “not”, which are represented in the graph by &,I, and Then for each effect, all combinations of causes that the effect depends on which will make the effect true are generated. By doing this, we identify the combinations of conditions that make different effects true. A test case is then generated for each combination of conditions, which make some effects true.

Example

Suppose that for a bank database there are two commands allowed:

Credit	acct-number	transaction-amount
Debit	acct-number	transaction-amount

The requirements are that if the command is credit and acct-number is valid, then the account is credited. If the command is debit, the acct-number is valid. and the transaction amount is valid then the amount is debited. If invalid command is given or the account number is invalid, or the debit amount is not valid, a suitable error message is generated. We identify the following causes and effects from the above requirements

Causes:

C1 Command is credit.

C2. Command is debit

C3. Account number is valid.

C4. Transaction amount is valid.

Effects:

E1. Print” invalid command”

E2. Print “invalid account number “.

E3. Print “Debit amount no valid”

E4. Debit account.

E5. Credit account.

The graph representation of Causes-Effect has shown in figure 6.2

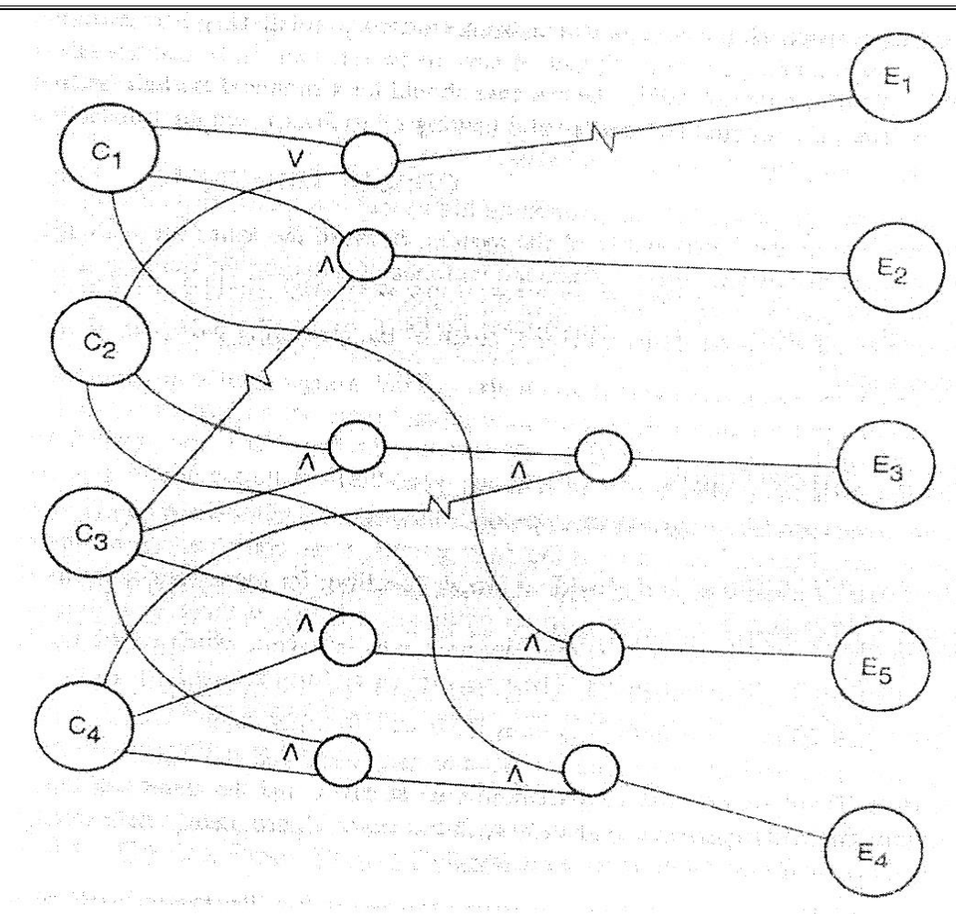


Figure 6.2 the Causes effect graph

7.6.2. Structural Testing

To test the structure of a program, structural testing methods are used. Several criterions have been proposed for structural testing. These criterions are precise and based on program structures. There are three different approaches to structural testing. They are

- Control flow based testing.
- Data flow based testing
- Mutation testing.

Control Flow-based Criteria

In this method, the control flow graph of a program is considered and coverage of various aspects of the graph is specified as criteria. A control flow graph G of a program P has set of nodes and edges. A node in this graph represents a block of statements that are always executed together. An edge (I, J) from node I to node J represents a possible transfer of control after executing he last statement of the block represented by node I to the first statement of the block represented by node J. A node corresponding to a block whose first statement is the start statement of P is called start node of G. Similarly, the node corresponding to a block whose last statement is an exit statement is called an exit node.

Now let us consider control flow-based criteria. The simplest coverage criteria are statement coverage, which requires that each statement of the program be executed at least once during testing. This is called all node criterions. This coverage criterion is not very strong and can

leave errors undetected. For example, if there is an if statement in the program without else part, the statement coverage criterion for this statement will be satisfied by a test case that evaluates the condition to true. No test case is needed that ensures that the condition in the if statement evaluates to false. This is a major problem because decisions in the programs are potential sources of errors.

Another coverage criterion is branch coverage, which requires that each edge in the control flow graph be traversed at least once during testing. In other words, branch coverage requires that each criterion in the program be evaluated to true and false values at least once during testing. Testing based on branch coverage criterion is known as branch testing. Problem with branch coverage comes if a decision has many conditions in it j(consisting of Boolean expression with Boolean operators "and" and "or"). In such a situation, a decision can be evaluated to true and false without actually exercising all conditions.

It has been observed that there are many errors whose presence is not detected by branch testing. This is because some errors are related to some combinations of branches and their presence is revealed by an execution that follows the path that includes those branches. Hence a more general coverage criterion which covers all the paths is required. This is called path coverage criterion and testing based on this criterion is called path testing. But the problem with this criterion is that programs that contain loops can have an infinite number of possible paths. Some methods have been suggested to solve this problem. One such method is to limit the number of paths.

Data flow-based Testing

In data flow-based testing, besides the control flow, information about where the variables are defined and where the definitions are used is also used to specify the test cases. The basic idea behind data flow-based testing is to make sure that during testing, the definitions of variables and their subsequent use is tested. For data flow-based testing, a definition-use graph for the program is first constructed from the control flow graph of the program. A statement in a node in the flow graph representing a block code has variable occurrences in it. A variable occurrence can be one of the following here types:

- Def represents the definition of the variable. Variables on the left hand side of an assignment statement are the one getting defined.
- C- use represents computational use of a variable. Any statement that uses the value of variables for computational purposes is said to be making use c-use of the variables. In an assignment statement, all variables on the right hand side have a c-use occurrence.
- P-use represents predicate use. These are all the occurrences of the variables in a predicate, which is used for transfer control.

Mutation Testing

Mutation testing is another type of structural testing and does not take path-based approach. Instead, it takes the program and creates many mutants of it by making simple changes in the program. The goal of testing is to make sure that during the course of testing, each mutant produces an output different from the output of the original program.

In mutation testing, first a set of mutants for a program under test P is prepared. Applying mutation operators on the text of P does this. The set of mutation operators depends on the language in which P is written. In general, a mutation operator makes a small unit change in the program to produce a mutant. Examples for mutant operators are: replace an arithmetic operator with some other arithmetic operator, change an array reference (say from I to J), replace a constant with another constant, and replace variable with some special value. Each application of a mutation operator results in one mutant.

Mutation testing of a program P proceeds as follows: First a set of test cases T is prepared by the tester, and P is tested by the set of test cases in T. If P fails, then T reveals some errors and they are corrected. If P does not fail, then it could mean that either the program P is correct or that P is not correct. But T is not sensitive enough to detect the fault in p. To rule out the latter possibility, the sensitivity of T is evaluated through mutation testing and more test cases are added to T until the set is considered sensitive enough for most faults.

7.7 LEVELS OF TESTING

Testing is used to detect faults introduced during specification and design stages as well as coding stages. Due to this, different levels of testing are used in the testing process. Each level of testing aims to test different aspects of the system. The basic levels of testing are.

- Unit testing
- Integration testing.
- System testing.
- Acceptance testing.

The relation of the faults introduced in different phases, and the different levels of testing are shown in figure 6.3.

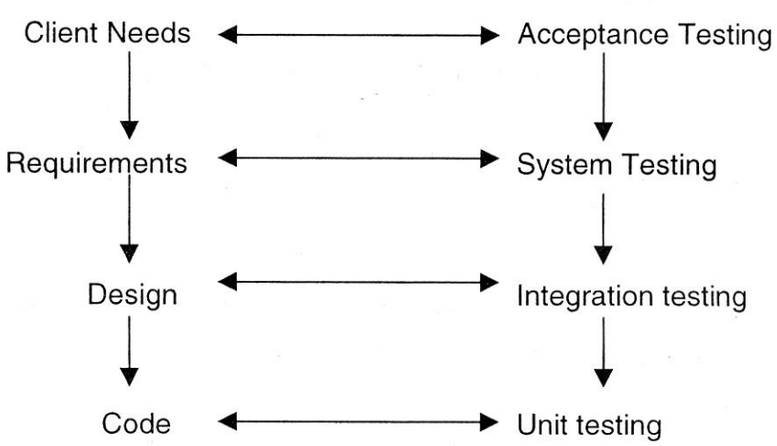


Figure 6.3 Levels of testing.

Unit testing

In this level, different modules are tested against the specification produced during design for the modules. Unit testing is essentially for verification f the code produced during the code phase. That is, the goal of this testing is to test the internal logic of the modules. Due to its close association with coding the coding phase is frequently called ”coding and unit testing” As the focus of this testing level is testing the code, structural testing is best suited for this level.

Integration testing

In this level, many unit-tested modules are combined into subsystems, which are then tested. The goal of this testing level is to see if the modules can be integrated properly. In other words the emphasis on testing the interfaces between the modules. This testing activity can be considered testing the design.

System testing

Here the entire software is tested. The reference document for this process is the requirements document. The goal is to see if the software meets its requirements. This is essentially validation exercise, and in many situations it is the only validation activity.

Acceptance testing.

Acceptance testing is performed using real data of the client to demonstrate that software is working satisfactorily. Testing here focuses on the external behavior of the system. Internal logic is not important for this testing. Hence, functional testing is performed at this level.

The above levels of testing are performed when the system is being built. We use another type of testing during the maintenance of the software. It is known as regression testing.

Regression testing is required when modification is made on the existing system. Software is said to be modified when we add one or mode modules to it or some of the components (or modules) deleted from it, Clearly, the modified software needs to be tested to make sure that it works properly.

7.8 TEST PLAN

In general, testing commences with a test plan and terminates with acceptance testing. A Test plan is a general document for the entire project that defines the scope, approach to be taken and the schedule of testing as will as identifies the test items for the entire testing process. It also contains information of the personnel responsible for the different activities of testing. The test planning can be done well before the actual testing commences and can be done in parallel with coding and design phases.

The inputs for forming the test plan are (1) Project plan, (2) requirements document, and (3) system design document. The project plan is needed to make sure that the test plan is consistent with the overall plan for the project and the testing schedule matches that of the project plan. The requirements document and the design document are the basic documents used for selecting the test units and deciding the approaches to be used during testing. A test plan should contain the following:

- Test unit specification
- Features to be tested
- Approach for testing.
- Test deliverables
- Schedule.
- Personnel Allocation

One of the most important activities of the test plan is to identify the test units A test unit is a set of one or more modules, together with associated data, that are from single computer program and that are object of testing. A test unit can occur at any level and can contain from a single module to the entire system.

Features to be tested include all software features and combinations of features that should be tested. A software feature is a software characteristic specified or implied by the requirements or design document. These may include functionality, Performance, design constraints, and attributes.

Testing deliverables should be specified in the test plan before the actual testing begins. Deliverables could be a list of test cases that were used, detailed results of testing, test summary report, log, and data about the code coverage.

The schedule specifies the amount of time and effort to be spent on different activities of testing and testing of different units that have been identified. Personnel allocation identifies the persons responsible for performing the different activities.

7.9 INTRODUCTION TO MAINTENANCE

Maintenance work is based on existing software, as compared to development work that creates new software. In other words, maintenance revolves around understanding the existing software and maintainers spend most of their time trying to understand the software they have to modify. Understanding the software means that understanding not only the code but also the related documents. During the modification of the software, the effect the change has to be clearly understood by the maintainer. To test whether those aspects of the system that are not supposed to be modified are operating as they were before modification, regression testing is done. In regression testing we use old test cases to test whether new errors have been introduced or not.

Thus, maintenance involves understanding the existing software, understanding the effect of change, making the changes to both code and documents, testing the new parts and retesting the old parts that were not changed. In order to make maintainer job easier, it is necessary to prepare some supporting documents during software development. The complexity of the maintenance task, coupled with the neglect of maintenance concerns during development, makes maintenance the most costly activity in the life of software product.

Maintenance is a set of software engineering activities that occur after software has been delivered to the customer and put into operation.

Maintenance activities can be divided into two types:

1. Modification-As the specifications of computer systems change, reflecting changes in the external world, so must the systems themselves.
2. Debugging-Removal of errors that should never have been there in the first place.

Software Maintenance Activities

Maintenance can be defined as four activities:

1. Corrective Maintenance-A process that includes diagnosis and corrective of errors.
2. Adaptive Maintenance- Activity that modifies software to properly interface with a changing environment (hardware and software).
3. Perfective Maintenance- Activity for adding new capabilities, modifying existing functions and making general enhancements.
4. Preventive Maintenance- Activity which changes software to improve future maintainability or reliability or to provide a better basis for future enhancements.

Distribution of maintenance activities

- Perfective:50%
- Adaptive:25%
- Corrective:21%
- Others(including Preventive):4%

Maintenance Costs

- Software organizations spend anywhere from 40 to 70 percent of all funds conducting maintenance.
- Reduction in overall software quality as a result of changes that introduce latent errors in the maintained software.

Types of Software Maintenance

In order for a software system to remain useful in its environment it may be necessary to carry out a wide range of maintenance activities upon it. Generally, there are three different categories of maintenance activities:

Corrective

Changes necessitated by actual errors in a system are termed corrective maintenance. A defect or "bug" can result from design errors, logic errors and coding errors. Design errors occur when for example changes made to the software are incorrect, incomplete, wrongly communicated or the change request misunderstood. Logic errors result from invalid tests and conclusions, incorrect implementation of design specification, faculty logic flow or incomplete test data. Coding errors are caused by incorrect implementation of detailed logic design and incorrect use of the source code logic. Defects are also caused by data processing errors and system performance errors. All these errors, sometimes called "residual errors" or "bugs" prevent the software from confirming to its agreed specification.

In the event of a system failure due to an error, actions are taken to restore operation of the software system. The approach here is to locate the original specifications in order to determine what the system was originally designed to do. However, due to pressure from management, maintenance personnel sometimes resort to emergency fixes known as "patching". The nature of this approach gives rise to a range of problems that include increased program complexity. Corrective maintenance has been estimated to account for 20% of all maintenance activities.

Adaptive

Any effort that is initiated as a result of changes in the environment in which a software system must operate is termed adaptive change. Adaptive change is a change driven by the need to accommodate modifications in the environment of the software system, without which the system would become increasingly less useful until it became obsolete.

The term environment in this context refers to all the conditions and influences which act from outside upon the system, for example business rules, government policies, work patterns,

software and hardware operating platforms. A change to the whole or part of this environment will warrant a corresponding modification of the software.

Unfortunately, with this type of maintenance the user does not see a direct change in the operation of the system, but the software maintainer must expend resources to effect the change. This task is estimated to consume about 25% of the total maintenance activity.

Perfective

The third widely accepted task is that of perfective maintenance. This is actually the most common type of maintenance ecompassing enhancements both to the function and the efficiency of the code and includes all changes, insertions, deletions modifications, extensions, and enhancements made to a system to meet the envolving and /or expanding needs of the user. A successful piece of software tends to be subjected to a succession of changes resulting in an increase in its requirements. This is based on the premise that as the software becomes useful, the users tend to experiments with new cases beyond the scope for which it was initially developed. Expansion in requirements can take the form of enhancement of existing system functionality or improvement in computational efficiency.

As the program continues to grow with each enhancement the system envolves from an average-sized program of average maintainability to a very large program that offers great resistance to modification. Perfective maintenance is by far the largest consumers of maintenance resources estimates of around 50% are not uncommon.

The categories of maintenance above were further defined in the 1993 IEEE Standard on Software Maintenance which goes on to define a fourth category.

Preventive

The long-term effect of corrective, adaptive and perfective change is expressed in Lehman's law of increasingly entropy:

As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it.

The IEEE defined preventive maintenance as "maintenance performed for the purpose of preventing problems before they occur". This is the process of changing software to improve its future maintainability or to provide a better basis for future enhancements.

The preventive change is usually initiated from within the maintenance organization within the maintenance organization with the intention of making programs easier to understand and hence facilitate future maintenance work. Preventive change does not usually give rise to a substantial increase in the baseline functionality.

Preventive maintenance is rare the reason being that other pressures tend to push it to the end of the queue. For instance, a demand may come to develop a new system that will improve the organizations competitiveness in the market. This will likely be seen as more desirable than spending time and money on a project that delivers no new function. Still, it is easy to see that if one considers the probability of a software unit needing change and the time pressures that are often present when the change is requested, it makes a lot of sense to anticipate change and to prepare accordingly.

PART A
MULTIPLE CHOICE QUESTIONS:
Application

1._____is a process of analysing a program with intended of finding errors.

A. Coding
B. Testing
C. Maintenance
D. Analyzing

2._____is a mechanism used to check correctness output of a program for test cases.

A. Coding
B. Testing
C. Maintainance
D. Testing oracles

3.In Top down testing approach what will stimulates the behavior of Sub Routines.

A. Driver
B. Stub
C. Causes
D. test Oracle

4.What are required in bottom up approach to set up a testing environment and invoke the modules for different testing cases.

A. test oracle
B. Cause
C. Drivers
D. stubs

5.Which term is used to refer to the discrepancy between computed, observed, or measured value and the true or specified value.

A. Detect errors
B. Failure
C. Errors
D. Fault

6._____is the inability of a system to perform a required function according to its specifications.

A. Failure
B. Testing
C. Errors
D. Fault

7._____are required as inputs to find out the presence of fault in a system.

A. Test cases
B. Test oracles
C. Testing principle
D. Testing approach

8.The ___testing procedure is exhaustive testing.

A. Structural testing
B. Functional testing
C. Unit testing
D. Integration testing

9.Which has been observed that programs that work correctly for a set of values in an equivalence class fail on some special values.

A. Equivalence class partitioning
B. Boundary value analysis
C. Cause-Effect Graphing
D. Functional testing

10.____is known as "Glass Box" testing.

A. Functional testing
B. Structural testing
C. Unit testing
D. System testing

11. ____is known as "Block Box" testing.

A. Functional testing
B. Structural testing
C. Unit testing
D. System testing

12.In control flow based testing, General coverage criteria based testing is called____

A. Path testing

B. Branch testing

C. Mutation testing

D. Acceptance testing

13. ____is used to ensure the definitions of variables and their subsequent use is tested.

A. Mutation testing

B. Data flow based testing

C. Control Flow-based testing

D. Structural testing

14.C-use variable represents

A. Code use of variable

B. Computational use of variable

C. Control use of variable

D. Constant use variable

15.Which variable occurrence is used for transfer of control in data flow based testing.

A. C-use variable

B. P-use variable

C. DEF variable

D. D-use variable

16. Which takes the program and creates many mutants of it by making simple changes in the program.

A. boundary value testing

B. Data flow based testing

C. Control Flow-based testing

D. Mutation testing

17.Which testing involves testing the interface between the modules to ensure modules are working properly in the system

A. Unit testing

B. Integration testing

C. System testing

D. Acceptance testing

18.A testing involving internal logic testing of modules w.r.t code is

A. Unit testing

B. Integration testing

C. System testing

D. Acceptance testing

19.Where the entire software is tested.

A. Unit testing

B. Integration testing

C. System testing

D. Acceptance testing

20. Which testing is performed using real data of the client to demonstrate that software is working satisfactorily.

A. Unit testing

B. Integration testing

C. System testing

D. Acceptance testing

21.A Testing during maintainence of software is_____

A. Unit testing

B. Maintainence testing

C. Regression testing

D. Adaptive testing

22.Changes in the environment in which software system must operate refers to

A. Corrective maintenance

B. Perfective maintenance

C. Adaptive maintenance

D. Preventive maintenance

23.Activity of adding new capability and modifying existing function is

A. Corrective maintenance

B. Perfective maintenance

C. Adaptive maintenance

D. Preventive maintenance

24.Which maintenance is largest consumer of maintenance resource

A. 1 Corrective maintenance

B. Perfective maintenance

C. Adaptive maintenance

D. Preventive maintenance

25.IEEE definition “maintenance performed for the purpose of preventing problems before they occur” is for_____

A. Corrective maintenance
B. Perfective maintenance
C. Adaptive maintenance
D. Preventive maintenance

PART B

APPLICATION

1. Explain the test oracle with the help of the diagram.
2. Briefly explain functional testing.
3. Explain equivalence class partitioning.
4. Explain boundary value analysis.
5. Explain the cause-effect graphing with the an example and diagram.
6. Briefly explain structural testing.

SKILL DEVELOPMENT

7. Justify the control flow based testing with suitable example.
8. Justify the data flow based testing with an example.
9. Express the view on Mutation Testing
10. Compare the different levels of testing.
11. Focus on test plan.
12. Summarise on maintenance? explain the software maintenance activities

VALUE- ADDED CHAPTERS

INTRODUCTION TO TESTING TOOLS

8.1 Overview of Win Runner

Win Runner is the most used Automated Software Testing Tool.

Main Features of **Win Runner** are

- Developed by Mercury Interactive
- Functionality testing tool
- Supports C/s and web technologies such as (VB, VC++, D2K, Java, HTML, Power Builder, Delphe, Cibell (ERP))
- To Support .net, xml, SAP, PeopleSoft, Oracle applications, Multimedia we can use QTP.
- Win runner run on Windows only.
- XRunner run only UNIX and Linux.
- Tool developed in C on VC++ environment.
- To automate our manual test win runner used TSL (Test Script language like c)

1.2 Silk Test

Silk Test is a tool for automated function and regression testing of enterprise applications. It was originally developed by Segue Software which was acquired by Borland in 2006. Borland was acquired by Micro Focus International in 2009.

Silk Test offers various clients:

- **Silk Test** Classic uses the domain specific 4Test language for automation scripting. It is an object oriented language similar to C++. It uses the concepts of classes, objects, and inheritance.
- **Silk4J** allows automation in Eclipse using Java as scripting language
- **Silk4Net** allows the same in Visual Studio using VB or C#
- **Silk Test** Workbench allows automation testing on a visual level (similar to former Test Partner) as well as using VB.Net as scripting language.

8.3 SQA Robot

Welcome to SQA Forums - The most popular Software Testing and Quality Assurance discussions site. With over 50 forums that cover almost every area in software testing, quality assurance and quality engineering. Here, you will also find a forum for every software test tool available like WinRunner, QuickTest Pro and LoadRunner by HP Mercury Interactive, IBM Rational Robot, TestPartner by Compuware and SilkTest by Borland Segue Software to name a few.

If you are looking for place to get help or support on any software testing tool, you've found the only place! Simply ask our **180,000+ Members** for almost anything, and you'll be surprised at the amount of help you can get here which you cannot get anywhere else. Looking for help on

WinRunner, LoadRunner, SilkTest, Robot, QARun, eTest, TestComplete and Webload? This is the place!

Important Notes:

- Registration is required to use this site. If you do not register, you will not be able to view or post to any forums.
- When registering, please make sure you use a valid email address and spell it correctly. When you register, your password is automatically emailed to you. If you do not provide a valid email address, you will not be able to login. Also note that we usually ban users who try to use invalid email addresses. If your password does not reach you after you have registered and you are sure you used a correct email address, notify the webmaster.
- Read your registration email carefully. The email includes forum rules and posting guidelines. If you violate the forum rules, you will get banned! Also read the FAQ Help if you have trouble logging in. Note that the username and password are case sensitive. Also note that if you use a public email service like Yahoo or Hotmail to make sure your account is NOT over quota. If it is, your registration email will bounce back and you will not get your password information.

8.4Load Runner

HP LoadRunner is an automated performance and test automation product from Hewlett-Packard for application load testing: examining system behaviour and performance, while generating actual load. HP acquired Load Runner as part of its acquisition of Mercury Interactive in November 2006.

A software testing tool, HP Load Runner works by creating virtual users who take the place of real users' operating client software, such as Internet Explorer, sending requests using the HTTP protocol to IIS or Apache web servers. HP Load Runner can simulate thousands of concurrent users to put the application through the rigors of real-life user loads, while collecting information from key infrastructure components (Web servers, database servers etc.) The results can then be analyzed in detail to explore the reasons for particular behaviour.

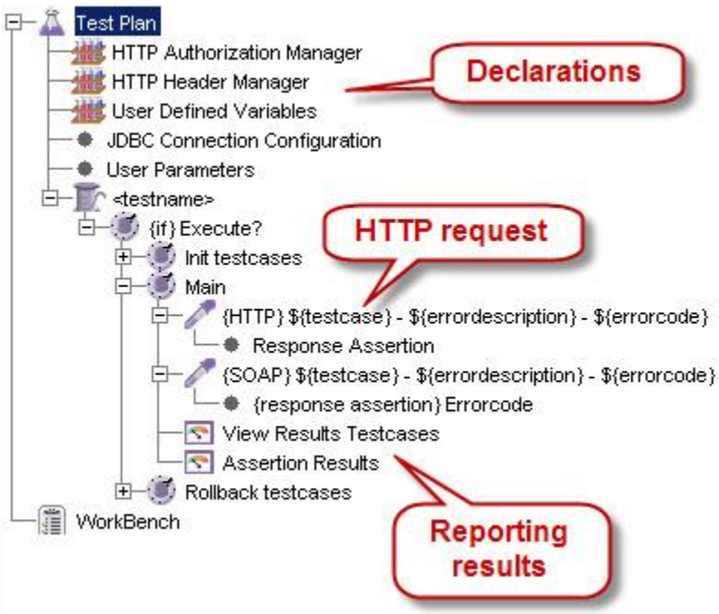
***Architecture*HP LoadRunner by default installs 3 icons on the Windows desktop:**

- **VuGen** (Virtual User Generator) for generating and editing scripts.
- **Controller** for composing scenarios which specify which load generators are used for which script, and for how long, etc. During runs the Controller receives real-time monitoring data and displays status.
- **Analysis** which assembles logs from various load generators and formats reports for visualization of run result data and monitoring data.

8**.5 JMeter and Text Director**

The principle of JMeter

The principle of JMeter is very simple. If you want to test e.g. a SOAP interface layer, all you basically need is the URL and SOAP request. Starting with that you can build your test plan. And this can be as fancy as you want. Using variables, counters, parameters, CSV files, loops, logs, etc. There are almost no limits in designing your test and making it as maintainable as possible.



Software Testing Techniques

Software Testing Techniques Programmers attempt to build s/w from an abstract concept to a tangible product. Engineers create a series of test cases that are intended to "demolish" the s/w that has been built. Testing is one step in s/w process that could be viewed as destructive rather than constructive

Testing objectives

• Testing is process of executing a program with the intent of finding an error

• A good test case is one that has a high probability of finding as an yet undiscovered error

• A successful test is one that uncovers an as yet undiscovered error

Software testability is simply how easily a computer program can be tested .Since testing is difficult, it pays to know what can be done to streamline it

Sometimes programmers are willing to do things that will help the testing process and a check list of possible design point features .There are certain metrics that could be used to

Measure testability in most of aspects, sometime testability is used to mean how adequately a Particular set of test will cover the product.

The Software Test Tool For Visual Testing

• BB TestAssistant records video, sound, webcam, keystrokes and mouse position to make a movie that's easy to view and distribute.

- BB TestAssistant records PC specification, mouse and keyboard inputs, and logging information generated by the target application:
  - Developers see exactly what happened - it gives an accurate and permanent record of events.
  - It integrates with applications under test.
  - Annotate movies with text, images, webcam recording and commentary to allow testers to comment on any problem.
  - Powerful editing and navigation lets you cut down recording to just show the issues.
  - Export to Flash, AVI, WMV, EXE or PPT formats, with full control over movie size and quality.
- BB TestAssistant records continuously to a set hard disk space or time limit, to catch intermittent errors
- Automatically records detailed PC configuration information to aid in problem diagnostics.
- API allows integration with target application and test tools.
- Records keystrokes and presents them in an easy to read, searchable user interface.
- Collects logfiles generated by external programs and includes them in movie report.

BB TestAssistant Software Test Tool - Benefits

- Ease of deployment and use - low training, no integration.
- Reduces technical support costs by speeding up reporting.
- Improves communications between test staff and developers - separates observations from inferences.
- Increase tester effectiveness by catching hard to reproduce bugs.
- Works in parallel with existing test strategies.
- BB TestAssistant also catches severe Windows application errors automatically.
- Visually explains the relationship between actions and observations.
- Reduces the need for form - filling.
- Eliminates reporting errors.
- Equally suitable for trained or untrained testers.

Simplified Communication For Accurate Software Testing

- BB TestAssistant works with project tracking systems to make bug reporting as simple as '˜Show-and-Tell' - allowing the developer to see exactly what the tester actually did.
- Designed specifically for software testing, BB TestAssistant records video, sound, webcam, keystrokes and mouse position to make a movie that's easy to view and distribute.
- BB TestAssistant is designed to work in conjunction with modern software test methodologies and heuristics - and it does so without impacting the performance of applications under test.

- In addition, BB TestAssistant can also capture log information generated by your own software and include it with the movie, giving a permanent record of all events - even errors that might not appear through the UI.
- Using BB TestAssistant eliminates the need for testers to remember exactly what they did before they hit a bug, so they can concentrate on performing the tests in a more informal, creative way.

Key Features:

- **Event log**: The Windows event log, external log files, key presses and mouse clicks are viewed side-by-side with the movie to show the status at any instant in time.

- **QA system integration**: BB TestAssistant integrates fully with current industry-standard QA systems such as JIRA and Bugzilla. The integration API is open, so it's easy to add support for a preferred QA system as well.
-
- **Record everything**: Everything the tester sees on their screen will be recorded by BB TestAssistant, even complex Windows Aero animations.
-
- **Editing**: Once the video has been recorded, you can edit it from within BB TestAssistant to include annotations and audio commentary (which can also be supplied in real-time). You also have access to other video editing options such as clipping, cropping and quality adjustment.
-
- **Skip to edits**: Each edit made by the tester acts as an anchor within the video that the developer can choose to skip to. This is particularly useful in long videos, or where the developer knows exactly which part he's looking for.
-
- **Projects**: This UI feature allows users to define projects. A project holds common configuration settings related to a single application or set of applications which a tester is working with.
-
- **Remove Inactive Periods**: This feature allows the user to identify and remove periods of inactivity within a movie.
-
- **Hide Other Processes**: This security / data protection feature allows the user to restrict BB TestAssistant to record only a specific set of processes.
-
- **Export to Word**: This feature allows a user to mark important points in a movie and add notes, then automatically produce a Word document containing screenshots of all these points, together with the notes.

Characteristics that lead to testable software

Operability: The better it works, the more efficiently it can be tested"

The system has few bugs (bugs adds analysis and reporting overhead to the test process)

No bugs block execution of tests

The product evolves in functional stages (allows simultaneous development and testing)

Observability:

What you see is what you test"

Distinct output is generated for each input

System states and variables are visible or queriable during execution

Past system states and variables are visible or queriable e.g.(transaction log)

All factor affecting output are visible

Incorrect output is easily identified

Internal errors are automatically detected through self testing mechanism

Internal errors are automatically reported

Source code is accessible

Testing is a process rather than a single activity. This process starts from test planning then designing test cases, preparing for execution and evaluating status till the test closure. So, we can divide the activities within the fundamental test process into the following basic steps:

1) Planning and Control
2) Analysis and Design
3) Implementation and Execution
4) Evaluating exit criteria and Reporting
5) Test Closure activities

1) Planning and Control:

Testplanning has following major tasks:

i. To determine the scope and risks and identify the objectives of testing.

ii. To determine the test approach.

iii. To implement the test policy and/or the **test strategy**. (Test strategy is an outline that describes the testing portion of the software development cycle. It is created to inform PM, testers and developers about some key issues of the testing process. This includes the testing objectives, method of testing, total time and resources required for the project and the testing environments.).

iv. To determine the required test resources like people, test environments, PCs, etc.

v. To schedule test analysis and design tasks, test implementation, execution and evaluation.

vi. To determine the **Exit criteria** we need to set criteria such as **Coverage criteria.** (Coverage criteria are the percentage of statements in the software that must be executed during testing. This will help us track whether we are completing test activities correctly. They will show us

which tasks and checks we must complete for a particular level of testing before we can say that testing is finished.)

Test control has the following major tasks:

i. To measure and analyze the results of reviews and testing.

ii. To monitor and document progress, test coverage and exit criteria.

iii. To provide information on testing.

iv. To initiate corrective actions.

v. To make decisions.

2) Analysis and Design:

Test analysis and Test Design has the following major tasks:

i. To review the **test basis.** (The test basis is the information we need in order to start the test analysis and create our own test cases. Basically it's a documentation on which test cases are based, such as requirements, design specifications, product risk analysis, architecture and interfaces. We can use the test basis documents to understand what the system should do once built.)

ii. To identify test conditions.

iii. To design the tests.

iv. To evaluate testability of the requirements and system.

v. To design the test environment set-up and identify and required infrastructure and tools.

3) Implementation and Execution:

During test implementation and execution, we take the test conditions into **test cases** and procedures and other **testware** such as scripts for automation, the test environment and any other test infrastructure. (Test cases is a set of conditions under which a tester will determine whether an application is working correctly or not.)

(Testware is a term for all utilities that serve in combination for testing a software like scripts, the test environment and any other test infrastructure for later reuse.)

Test implementation has the following major task:

i. To develop and prioritize our test cases by using techniques and create **test data** for those tests. (In order to test a software application you need to enter some data for testing most of the features. Any such specifically identified data which is used in tests is known as test data.) We also write some instructions for carrying out the tests which is known as **test procedures.** We may also need to automate some tests using **test harness** and automated tests scripts. (A test harness is a collection of software and test data for testing a program unit by running it under different conditions and monitoring its behavior and outputs.)

ii. To create test suites from the test cases for efficient test execution. (Test suite is a collection of test cases that are used to test a software program to show that it has some specified set of behaviours. A test suite often contains detailed instructions and information for each collection of test cases on the system configuration to be used during testing. Test suites are used to group similar test cases together.)

iii. To implement and verify the environment.

Test execution has the following major task:

i. To execute test suites and individual test cases following the test procedures.

ii. To re-execute the tests that previously failed in order to confirm a fix. This is known as **confirmation testing or re-testing.**

iii. To log the outcome of the test execution and record the identities and versions of the software under tests. The **test log** is used for the audit trial. (A test log is nothing but, what are the test cases that we executed, in what order we executed, who executed that test cases and what is the status of the test case (pass/fail). These descriptions are documented and called as test log.).

iv. To Compare actual results with expected results.

v. Where there are differences between actual and expected results, it report discrepancies as Incidents.

4) Evaluating Exit criteria and Reporting:

Based on the risk assessment of the project we will set the criteria for each test level against which we will measure the "enough testing". These criteria vary from project to project and are known as **exit criteria**.

Exit criteria come into picture, when:

— Maximum test cases are executed with certain pass percentage.

— Bug rate falls below certain level.

— When achieved the deadlines.

Evaluating exit criteria has the following major tasks:

i. To check the test logs against the exit criteria specified in test planning.

ii. To assess if more test are needed or if the exit criteria specified should be changed.

iii. To write a test summary report for stakeholders.

5) Test Closure activities:

Test closure activities are done when software is delivered. The testing can be closed for the other reasons also like:

- When all the information has been gathered which are needed for the testing.
- When a project is cancelled.
- When some target is achieved.
- When a maintenance release or update is done.

Test closure activities have the following major tasks:

i. To check which planned deliverables are actually delivered and to ensure that all incident reports have been resolved.

ii. To finalize and archive testware such as scripts, test environments, etc. for later reuse.

iii. To handover the testware to the maintenance organization. They will give support to the software.

iv To evaluate how the testing went and learn lessons for future releases and projects.

Session2: Test case design

- There is only one rule in designing test case cover all features but do not make too many test cases

- The highest likelihood of finding the most errors with a minimum amount of time and Effort

The Software Test Tool For Visual Testing

- BB TestAssistant records video, sound, webcam, keystrokes and mouse position to make a movie that's easy to view and distribute.
- BB TestAssistant records PC specification, mouse and keyboard inputs, and logging information generated by the target application:
  - Developers see exactly what happened - it gives an accurate and permanent record of events.
  - It integrates with applications under test.
  - Annotate movies with text, images, webcam recording and commentary to allow testers to comment on any problem.
  - Powerful editing and navigation lets you cut down recording to just show the issues.
  - Export to Flash, AVI, WMV, EXE or PPT formats, with full control over movie size and quality.
- BB TestAssistant records continuously to a set hard disk space or time limit, to catch intermittent errors
- Automatically records detailed PC configuration information to aid in problem diagnostics.
- API allows integration with target application and test tools.
- Records keystrokes and presents them in an easy to read, searchable user interface.
- Collects logfiles generated by external programs and includes them in movie report.

BB TestAssistant Software Test Tool - Benefits

- Ease of deployment and use - low training, no integration.
- Reduces technical support costs by speeding up reporting.
- Improves communications between test staff and developers - separates observations from inferences.
- Increase tester effectiveness by catching hard to reproduce bugs.
- Works in parallel with existing test strategies.
- BB TestAssistant also catches severe Windows application errors automatically.
- Visually explains the relationship between actions and observations.
- Reduces the need for form - filling.
- Eliminates reporting errors.
- Equally suitable for trained or untrained testers.

Simplified Communication For Accurate Software Testing

- BB TestAssistant works with project tracking systems to make bug reporting as simple as '˜Show-and-Tell' - allowing the developer to see exactly what the tester actually did.

- Designed specifically for software testing, BB TestAssistant records video, sound, webcam, keystrokes and mouse position to make a movie that's easy to view and distribute.
- BB TestAssistant is designed to work in conjunction with modern software test methodologies and heuristics - and it does so without impacting the performance of applications under test.
- In addition, BB TestAssistant can also capture log information generated by your own software and include it with the movie, giving a permanent record of all events - even errors that might not appear through the UI.
- Using BB TestAssistant eliminates the need for testers to remember exactly what they did before they hit a bug, so they can concentrate on performing the tests in a more informal, creative way.

Key Features:

- **Event log**: The Windows event log, external log files, key presses and mouse clicks are viewed side-by-side with the movie to show the status at any instant in time.
- **QA system integration**: BB TestAssistant integrates fully with current industry-standard QA systems such as JIRA and Bugzilla. The integration API is open, so it's easy to add support for a preferred QA system as well.
- **Record everything**: Everything the tester sees on their screen will be recorded by BB TestAssistant, even complex Windows Aero animations.
- **Editing**: Once the video has been recorded, you can edit it from within BB TestAssistant to include annotations and audio commentary (which can also be supplied in real-time). You also have access to other video editing options such as clipping, cropping and quality adjustment.
- **Skip to edits**: Each edit made by the tester acts as an anchor within the video that the developer can choose to skip to. This is particularly useful in long videos, or where the developer knows exactly which part he's looking for.
- **Projects**: This UI feature allows users to define projects. A project holds common configuration settings related to a single application or set of applications which a tester is working with.
- **Remove Inactive Periods**: This feature allows the user to identify and remove periods of inactivity within a movie.
- **Hide Other Processes**: This security / data protection feature allows the user to restrict BB TestAssistant to record only a specific set of processes.
- **Export to Word**: This feature allows a user to mark important points in a movie and add notes, then automatically produce a Word document containing screenshots of all these points, together with the notes.

Session3: Two Unit Testing Techniques

Black box testing

Knowing the specified function that a product has been designed to perform, test to see if that function is fully operational and error free

Includes tests that are conducted at the software interface

Not concerned with internal logical structure of the software

White box testing

Knowing the internal workings of a product, test that all internal operations are performed according to specifications and all internal components have been exercised

Involves tests that concentrate on close examination of procedural detail

Logical paths through the software are tested

Test cases exercise specific sets of conditions and loops

White box Testing

Uses the control structure part of component level design to derive the test cases

These test cases

Guarantee that all independent paths within a module have been exercised at least once

Exercise all logical decisions on their true and false sides

Execute all loops at their boundaries and within their operational bounds

Exercise internal data structures to ensure their validity.

Logical errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed

We often believe that a logical path is not likely to be executed when, in fact , it may be executed on a regular basis

Typographical errors are random.

Black box testing treats the system as a **"black-box"**, so it doesn't explicitly use Knowledge of the internal structure or code. Or in other words the Test engineer need not know the internal working of the "Black box" or application.

Main focus in black box testing is on functionality of the system as a whole. The term **'behavioral testing'** is also used for black box testing and white box testing is also sometimes called **'structural testing'**. Behavioral test design is slightly different from black-box test design because the use of internal knowledge isn't strictly forbidden, but it's still discouraged.

Each testing method has its own advantages and disadvantages. There are some bugs that cannot be found using only black box or only white box. Majority of the applicationa are tested by black box testing method. We need to cover majority of test cases so that most of the bugs will get discovered by blackbox testing.

Black box testing occurs throughout the software development and Testing life cycle i.e in Unit, Integration, System, Acceptance and regression testing stages.

Tools used for Black Box testing

Black box testing tools are mainly record and playback tools. These tools are used for regression testing that to check whether new build has created any bug in previous working application functionality. These record and playback tools records test cases in the form of some scripts like TSL, VB script, Java script, Perl.

Advantages of Black Box Testing

– Tester can be non-technical.

– Used to verify contradictions in actual system and the specifications.

– Test cases can be designed as soon as the functional specifications are complete

Disadvantages of Black Box Testing

– The test inputs needs to be from large sample space.

– It is difficult to identify all possible inputs in limited testing time. So writing test cases is slow and difficult

– Chances of having unidentified paths during this testing

White box testing traditionally refers to the use of program source code as a test basis, that is, as the basis for designing tests and test cases. (IEEE standards define a test as "a set of one or more test cases.") A looser definition says that "white box testing is based on internal structures of the software", but it is very unclear what kinds of "internal structure" are covered by this. Some authorities, for example, include user-facing menus and even business processes. The term "gray box testing" is now in widespread use for internal software structures that are not actually source code - for example, class hierarchies or module call trees. In the following discussion, "white box testing" is held to be synonymous with "code-based testing."

Those two terms are opposed to "requirements-based" or "specification-based" testing, also known as black box testing. The implication there is that you can't see the inner workings of a black-painted box; and in fact, you don't need to see the inner workings to test whether a given set of inputs results in the required or specified set of outputs. But if you do need to test the inner workings, painting the box white would leave them just as invisible as when the box was black, so the terms "clear box testing" and "glass box testing" are also used. (The word "analysis" is sometimes used in place of "testing.")

White-box testing usually involves tracing possible execution paths through the code and working out what input values would force the execution of those paths. Quite simple techniques exist by which the tester (usually the developer who wrote the code) can work out the smallest number of paths necessary to test, or "cover," all the code. Some types of static analysis tool will do the same job, more quickly and more reliably.

White box testing, on its own, cannot identify problems caused by mismatches between the actual requirements or specification and the code as implemented but it can help identify some types of design weaknesses in the code. Examples include control flow problems (e.g., closed or infinite loops or unreachable code), and data flow problems (e.g., trying to use a variable which has no defined value). Static code analysis (by a tool) may also find these sorts of problems, but doesn't help the tester/developer understand the code to the same degree that personally designing white-box test cases does.

White box testing is usually associated with component testing (i.e., unit testing), and with the concept that, at minimum, 100% of the component's code statements should be tested before the component is released for use ("100% code coverage"). But it may be dangerous to design white box test cases simply to achieve such a standard. This is not only because it will miss disconnects

between code and specifications, but because the test cases are often artificial and unrepresentative of how the component will actually be used. The test cases may even be impossible to execute (corresponding to "unachievable" or "infeasible" paths. A tool will avoid generating these).

A better approach may be to design enough test cases to cover all requirements specified for the component (black box testing), then use a code coverage monitor to record how much of the code is actually covered (executed) when the test cases are run. Often, the code will fall into two parts: code statements that correspond directly to the requirements and get covered by the black box test cases; and code statements that deal with aspects of the target execution environment, which may not be explicitly specified but which the developer is expected to provide. This second type of code often provides error handling for exceptions thrown by hardware or software conditions (e.g., by a database management system), and often goes untested - which may cause severe problems when such an exception occurs in live use. The developer would use white box techniques to work out how to drive the testing of those statements not covered by the black box test cases.

Session4: Basic path testing

•White box testing technique proposed by Tom McCabe

• Enables the test case designer to derive a logical complexity measure of a procedural design

• Uses this measure as a guide for defining a basis set of execution paths

•Test cases derived to exercise the basis set are guaranteed to execute every statement in the Program at least one time during testing

Path Testing: Independent Paths

This post follows on from the previous two in this series (theory and path coverage). A key theme of these posts is that of modeling paths through an application and then using tests to exercise those paths. What we've seen so far is that while many path segments are covered repeatedly, others may only be touched once. You most likely won't have time to test them all. Along with that, some paths will be more data-dependent than others. So a selection process is critical. Here I'll talk more about that selection process.

Using the terminology I've already introduced, a path through a program is a node and edge sequence from the starting node to a terminal (ending) node of the control flow graph of a program. That's how the academics will word it anyway and while it is accurate, it can be hard to see how to apply it. In this post, I'm going to try to give an example.

Before going on I'll state that writing tests to cover all the paths of a typical application is usually impractical, at least depending on the nature of the application and the various states it can be in based on data that is input, processed and output. This inherent complexity is why testing based on path analysis usually does not require coverage of *all paths* but instead focuses on coverage of the *linearly independent paths*.

Testing Independent Paths

I talk about this in the other posts, but to briefly recap, a linearly independent path is any path through the application that introduces at least one new node that is not included in any other

linearly independent path. That's easy enough, right? But now consider this: if a path has one new node compared to all other linearly independent paths, then that path is also linearly independent. The reason for this is because any path having a new node automatically implies that it has a new edge. This means that a path that is a sub-path of another path is *not* considered to be a linearly independent path.

It always feels like a challenge to make this stuff understandable. Before I move on to an example where I'll show you a way to calculate independent paths (which I hope makes it a lot clearer!), you might be saying, *"What if I don't have access to the source code like you've been showing in these posts? What if I'm not performing code-based unit testing?"*

you can still do path analysis in terms of those aspects of code that filter up through a user interface. In the example of either the Greatest Common Divisor or Maximum Integer, which I showed in the last post, if there was a user interface, such as a form, that allowed a user to type in values to get a value back, you could do your path analysis based on that. Think to any web site you've used. Certainly you would be able to figure out and model paths through it even without having access to the logic behind it.

So now you might say: *"Well, okay, so if that's the case, why am I even bothering to read these posts about path coverage and path analysis if I can just figure it out by looking at its user interface?"*

Sometimes the user interface is not always amenable to that kind of analysis. A GUI might be (to some extent) but consider a web service or an algorithm engine. Even in the case of a GUI (whether desktop client, browser application, or mobile app), it's very possible that certain particular functions are never called directly by any aspect of the user interface. I worked on a hedge fund application and a clinical trial application and, in both cases, there was vast amounts of stuff going on below the GUI that would never have been covered in path analysis if those paths were considered solely from the GUI.

So this is where communication between testers and developers can take place to make sure that effective test coverage is achieved. This is also where a distinction can be made between coverage that's done at the unit (i.e., code) level and coverage that's done at the integration level or acceptance levels. As many tester roles have shifted into SDETs (Software Development Engineer in Test) or some variation thereof, the ability to look into code becomes not just possible but also part of your job.

Calculate the Independent Paths

What I want to show you now is how you can calculate the number of linearly independent paths through *any* structured system. That's the key to seeing how to apply this yourself on your own applications. My goal here is to come up with a way to find the maximum number of linearly independent paths in a given part of a hypothetical application that I'm testing. That will allow me to structure test cases based on that maximum number.

It's important to understand the idea of paths, so let's go through that idea really quick with an isolated example. Say you have these paths:

- **path 1:** 1-11

- **path 2:** 1-2-3-4-5-10-1-11
- **path 3:** 1-2-3-6-8-9-10-1-11
- **path 4:** 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge — meaning, a line to new nodes on the path.

Now consider this path:

- **path 5:** 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

Path 5 is *not* considered to be an independent path because it's simply a combination of the already specified paths (paths 2 and 3) and does not traverse any new edges.

Paths 1, 2, 3, and 4 constitute a *basis set* for a possible control flow graph. That is, if tests can be designed to force execution of those initial four paths, which make up a basis set, then *every* statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides.

Path testing is sometimes referred to as basis path testing and now you know why. A basis set is a set of linearly independent test paths. Any path through the control flow graph can be formed as a combination of paths in the basis set. I should note that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

Reg.No

SRINIVAS UNIVERSITY
FIFTH SEMESTER BCA DEGREE EXAMINATION NOVEMBER-2019
SOFTWARE PROJECT MANAGEMENT
PAPER CODE:17BCASD54/17BCACC54

Time: 2 Hours Max. Marks: 50

Instructions: Answer any 10 Questions from PART-A and One Full Questions from Each UNIT in PART-B

PART – A (10x1 = 10)

1.
a. IEEE defines _____ is a systematic approach to the development, operation, maintenance and requirement of the software

A. SOFTWARE Engineering
B. B requirement specification
C. C. coding
D. D. SRS

b. Blocking states is found in _____ model

E. Waterfall
F. Spiral
G. Iterative enhancement
H. Prototype

c. The system defined in multiple point of view is referred as

E. state
F. function
G. Object
H. Projection

d. An Evolutionary prototype leads to _____model

A. Prototype model
B. Waterfall model
C. Iterative enhancement model
D. Spiral model

e. The author of SRS document is____

A. developer
B. client
C. end user
D. analyst

f. An _____of a component describes the external behavior of that component without bothering with the internal details that produce the behavior.

E. Coupling
F. Cohesion
G. Abstraction
H. Factoring

g. The main module is a _____module.

A. composite
B. Co-ordinate
C. Transform
D. Output

h. ___is the process of decomposing a module so that the bulk of its work is done by its subordinates.

A. Factoring
B. *abstraction*
C. modularization
D. coupling

i. What is the aim of the design review.

A. Detecting the errors
B. Modifying the design
C. Restating the problem
D. Producing a Structured chart

j. Which is a manual method of verification in detailed design phase.

A. Desing walkthrough
B. Critical design review
C. Consistency checkers
D. Path condition

k. In consistency checker _____ can not be generated.

A. errors
B. Anomaly
C. output
D. Execution code

l. In Top down testing approach what will stimulates the behavior of Sub Routines.

A. Driver
B. Stub
C. Causes
D. test Oracle

PART-B

UNIT-I

2. a. List out the limitations of waterfall model

b. Explain the phases involved in project management process

(4 + 4)

OR

3. a. Explain different phases of development process

b. Write a note on capability maturity Model

(4 + 4)

UNIT II

4. a. Explain the characteristics of SRS

b. Write a note on common errors occurred during requirement phase

(4 + 4)

OR

5 a. Briefly explain the phases of requirement process with diagram

b. Explain briefly structured analysis technique

(4 + 4)

UNIT-III

6 a. What is cohesion? mention different levels of cohesion and explain any 2.

b.What is Factoring ?explain First level factoring with example

(4 + 4)

OR

7 a. Explain the lowest and strongest level of cohesion with suitable example

b.Write a note on verification in the detailed design phase.

(4 + 4)

UNIT-IV

8.a. What is data flow anomalies? explain with example

b.Explain any four programming style.

(4 + 4)

OR

9 a. Explain the symbolic execution and execution tree with an example

b. Explain static analysis and its uses. (4 + 4)

UNIT –V

10a. Explain equivalence class partitioning.

b. Explain different levels of testing (4 + 4)

OR

11 a. Explain the test oracle with the help of the diagram.

b. What is maintainance?explain the software maintainance activities (4 + 4)

****