SRINIVAS UNIVERSITY

INSTITUTE OF COMPUTER SCIENCE & INFORMATION SCIENCES

CITY CAMPUS, PANDESHWAR, MANGALORE-575 001

BACKGROUND STUDY MATERIAL

SOFTWARE ENGINEERING

B.C.A - V SEMESTER

Compiled by
Faculty

SOFTWARE ENGINEERING

CONTENTS

MODULE-1

MODULE-2

3 . Software Requirement Analysis and Specification

MODULE-3

4. Function Oriented Design

MODULE-4

5. Detailed Design

Detailed Design

PDL

Logic/Algorithm Design

Assignment 5

6. Coding

Programming Practice

Structured Programming

Programming Style

Internal Documentation

Verification

Code Reading

Assignment 6

MODULE-5

7. Software Testing and Maintenance

Introduction

Test Oracles

Type of Testing

Functional Testing

Structural Testing

Levels of Testing

Introduction to Maintenance

Software Maintenance Activities

Maintenance Costs

Assignment 7

SOFTWARE ENGINEERING

Total hours:30H

MODULE - I

6 hrs

Introduction

Session 1: The Software Problem, Software Engineering Problem, The Software Engineering Approach

Session 2: Phased Development Process, Software Process, Processes, Projects and Products

Session 3: Software Development Process, A Process Step Specification

Session 4: Waterfall Model, Prototype Model

Session 5: Iterative Enhancement Model, Spiral model

Session 6: Process Management Process, SCM

MODULE – II

6 hrs

Software Requirements Specification

Session 7: Software Requirements, Need for SRS

Session 8: Requirement process, Structured Analysis

Session 9: Data Flow Diagrams and Data Dictionary

Session 10: Prototyping, Requirements Specification

Session 11: Characteristics of an SRS, Components of an SRS

Session 12: Specification Languages, Regular Expressions, Structure of a Requirements Document

MODULE – III

6 hrs

Function Oriented Design

Session 13: Introduction, Design principles

Session 14: Problem partitioning and Hierarchy

Session 15 Abstraction, Module level Concepts

Session 16: Coupling, Cohesion

Session 17: Structured Design Methodology.

Session 18: Design Heuristics, Design Validation/Verification

MODULE – IV

6 hrs

Detailed Design

Session 19: Detailed design, PDL

Session 20: Logic/Algorithm Design

Session 21: Programming Practice

Session 22: Structured Programming, Programming Style

Session 23: Internal Documentation

Session 24: Verification, Code Reading

MODULE – V

6 hrs

Software Testing and Maintenance

Session 25: Introduction, Test Oracles, Type of Testing

Session 26: Functional Testing

Session 27: Structural Testing

Session 28: Levels of Testing

Session 29: Introduction to Maintenance

Session 30: Software Maintenance Activities, Maintenance Costs

TEXTBOOKS:

1. Integrated Approach to Software Engineering by Pankaj Jalote.

Scheme of Evaluation:
The paper carries 50 marks out of which 25 marks will be allotted to external examination and 25 marks will be allotted to the internal assessment.
Internal assessment marks will be calculated as follows

1. Performance in 3 IA examinations will be converted out of 15 marks
2. Attendance 5 marks
3. Assignment 5 marks.

   Total 25 marks.

External examination marks will be as follows:

1.	1 marks 5 questions	1 X 5 = 5 marks.
2.	One question out of two questions in each unit carries	4 X 5 = 20 marks
		Total 25 marks.

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

****

MODULE - I
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.

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

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.

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.

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.**

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".

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.

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.

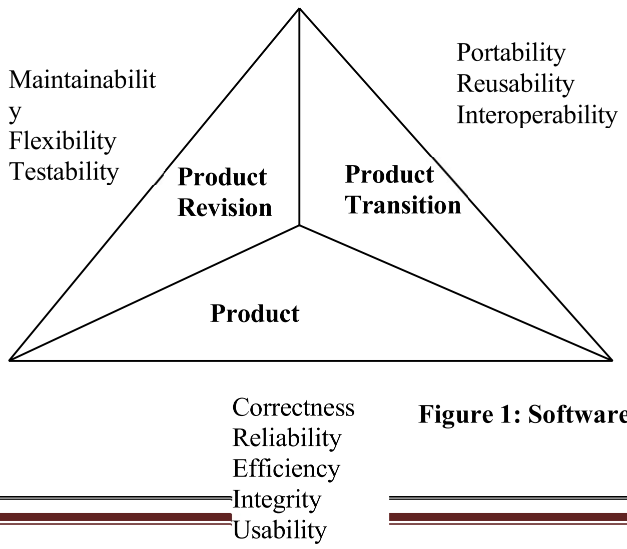


Figure 1: Software Quality Factors

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.

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.

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.

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.

CHAPTER - 2
SOFTWARE PROCESSES

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.

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.

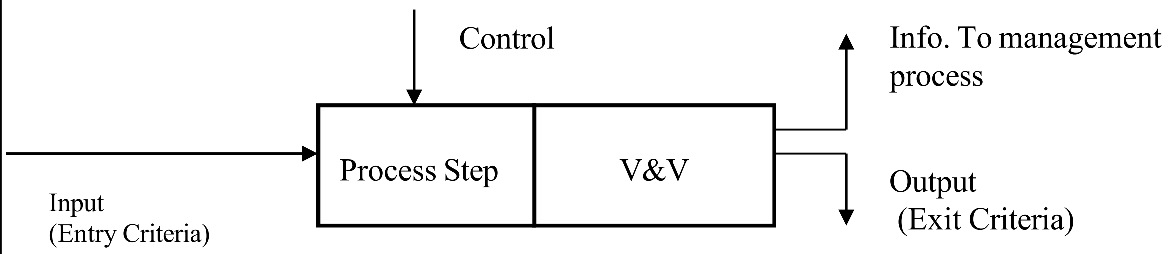
Software Development Process

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.

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.

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~~

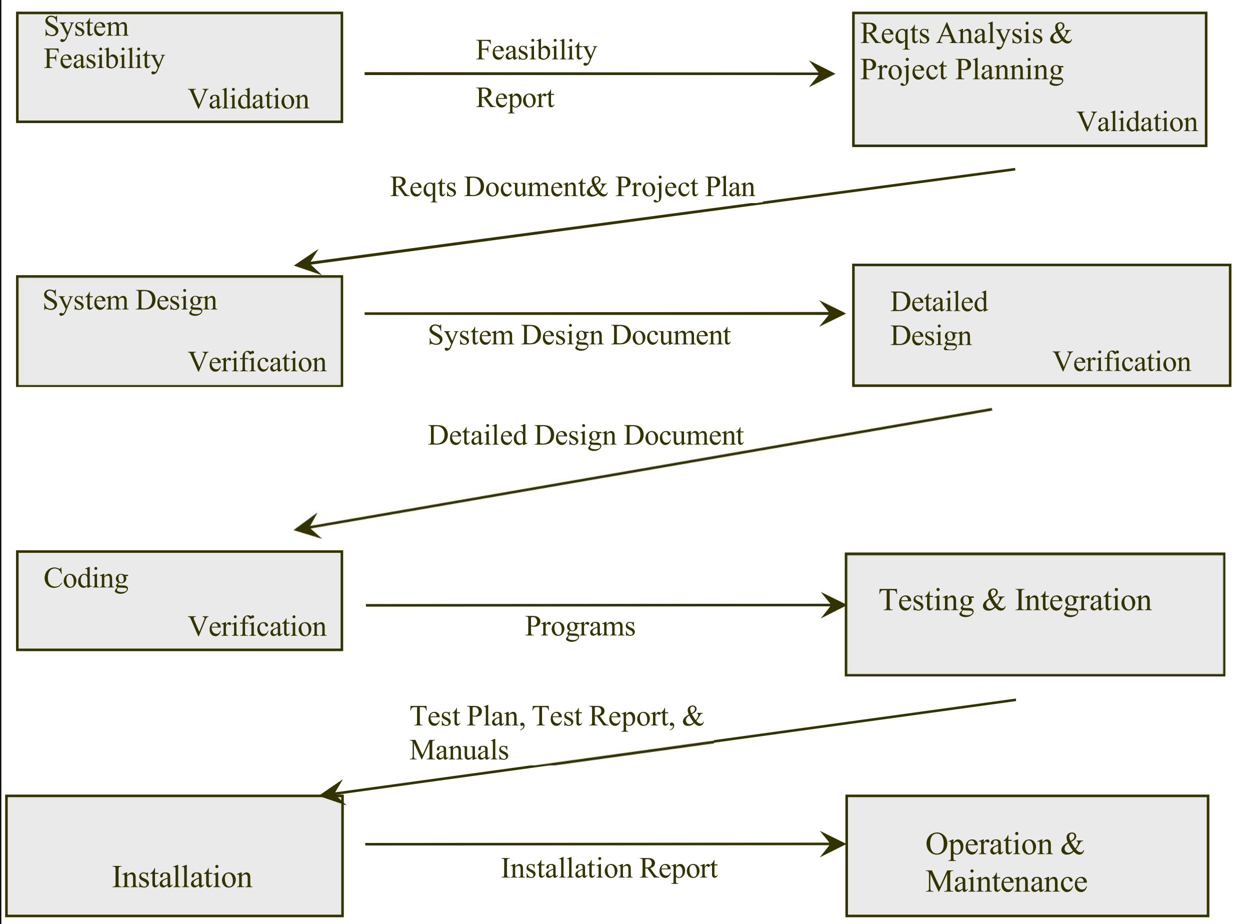
feasibility of a project, the requirement analysis and project planning begins. The design starts after the requirement analysis is complete and the coding begins after the design is complete, once the programming is complete, the code is integrated and testing is done. On successful completion of testing, 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

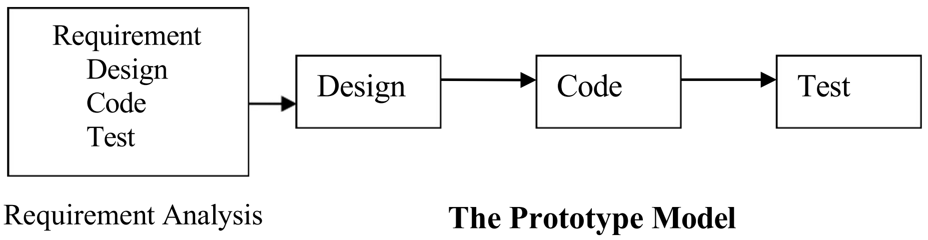
complete the dependent task. The time spent in waiting can exceed the time spent in productive work.

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

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.



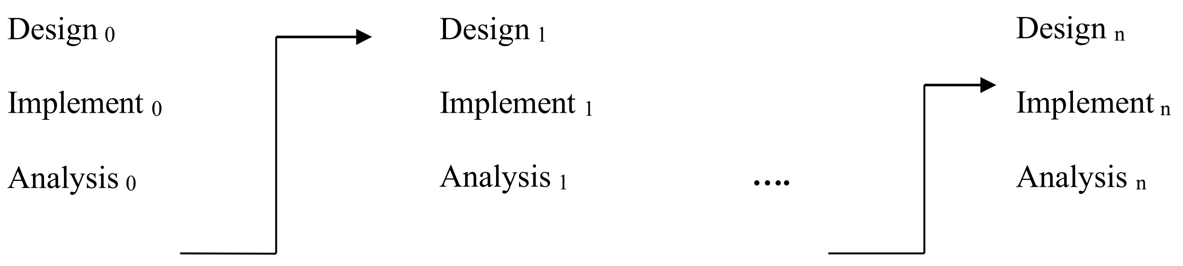
The Prototype Model

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.

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.

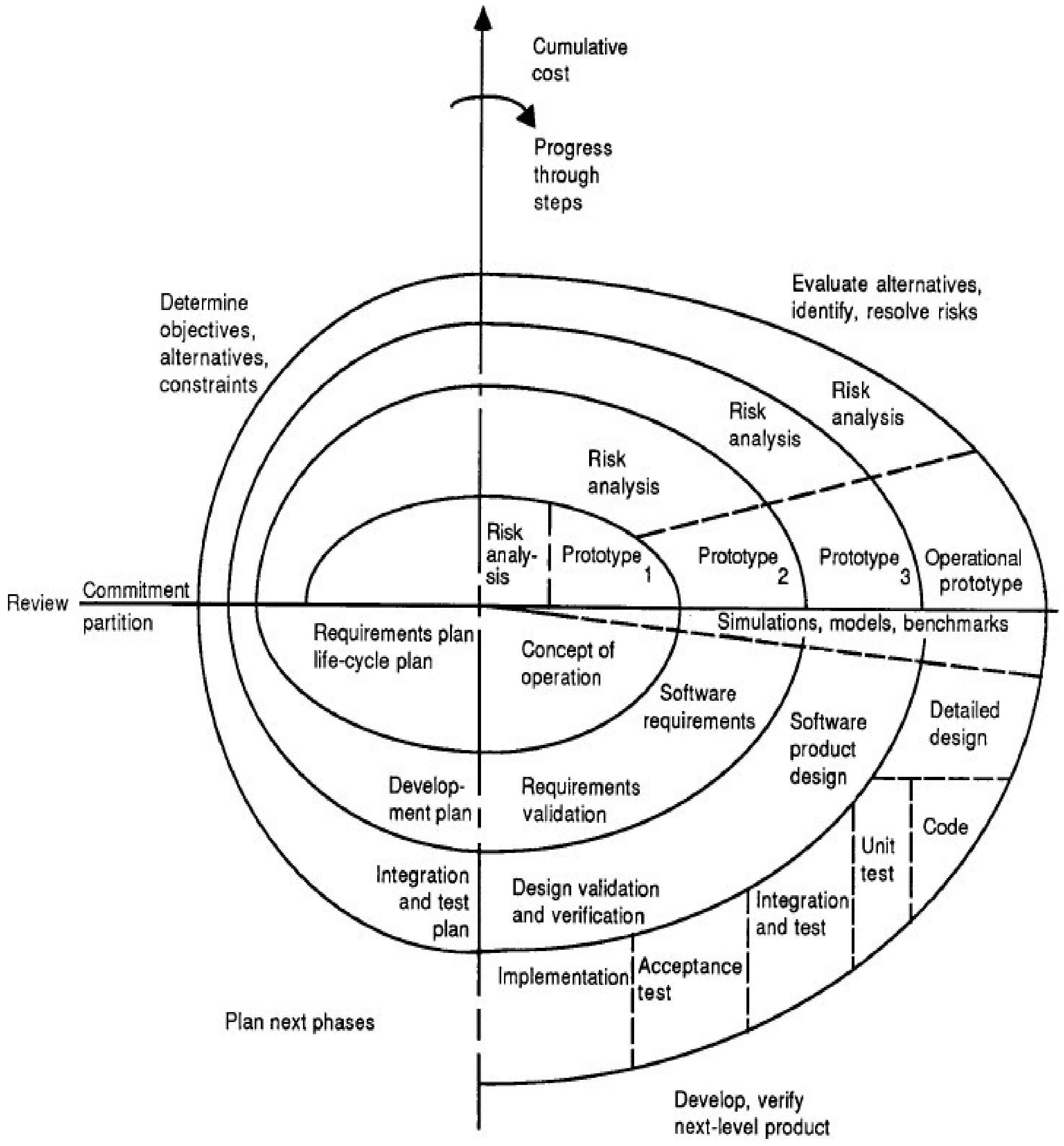
The first version contains some capability. Based on the feedback from the users and experience with the current version, a list of additional features is generated. And then more features are added to the next versions. This type of process model will be helpful only when the system development can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

Disadvantage:

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

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.

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.

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.

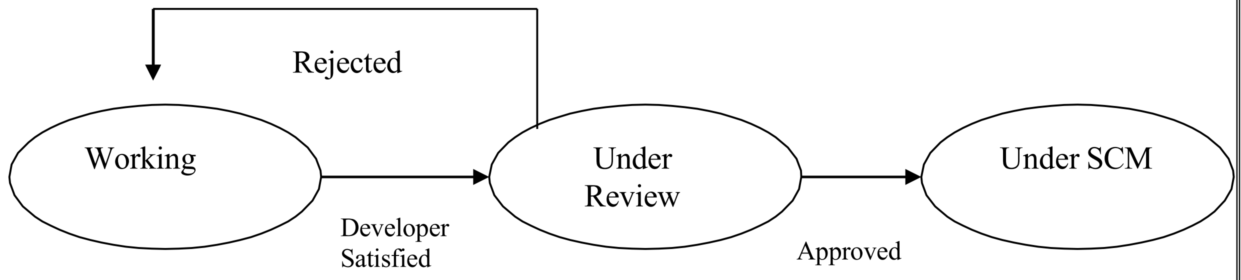
Though the goal of SCM is to control the establishment and changes to these baselines, treating each baseline as a single entity for the change is undesirable, because the change may be limited to a very small portion of the baseline. For this reason, a baseline can consist of many software configuration 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.

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.

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.

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.
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.
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. **High quality software product**

4. Quote from the memory that ___________is the effort required to couple system with other system.
A. Maintainability
B. **Inter-operability**
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. **Portability**
D. Reliability

6. State that ___________is the effort required to locate and fix the errors in the program.
A. **Maintainability**
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.
A. Waterfall
B. **Spiral**
C. Iterative enhancement
D. Prototype

9. Blocking states is found in ________model.
A. **Waterfall**
B. Spiral
C. Iterative enhancement
D. Prototype

10. Freezing of requirement before development process is a limitation in ________.
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)

Remembering

1. Define Software Engineering. Explain various problems faced in Software Engineering.
2. Mention the Quality attributes of Software Engineering.
3. Explain different phases of the development process.
4. State and explain the working of the waterfall model with the help of a diagram.
5. List out the limitations of the waterfall model.

Understanding

6. Explain the working of the iterative enhancement model.
7. Describe the spiral model with the help of diagram.
8. Describe the SCM life cycle of an item.
9. Explain various activities of Software Configuration Management Process

MODULE - II

CHAPTER - 3

SOFTWARE REQUIREMENT ANALYSIS AND SPECIFICATION

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.

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.

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.

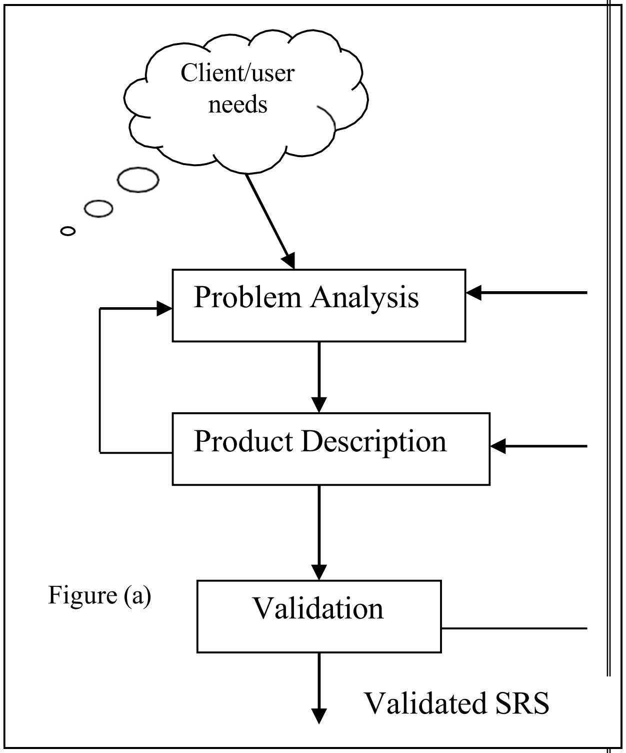


Figure (a)

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.

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.

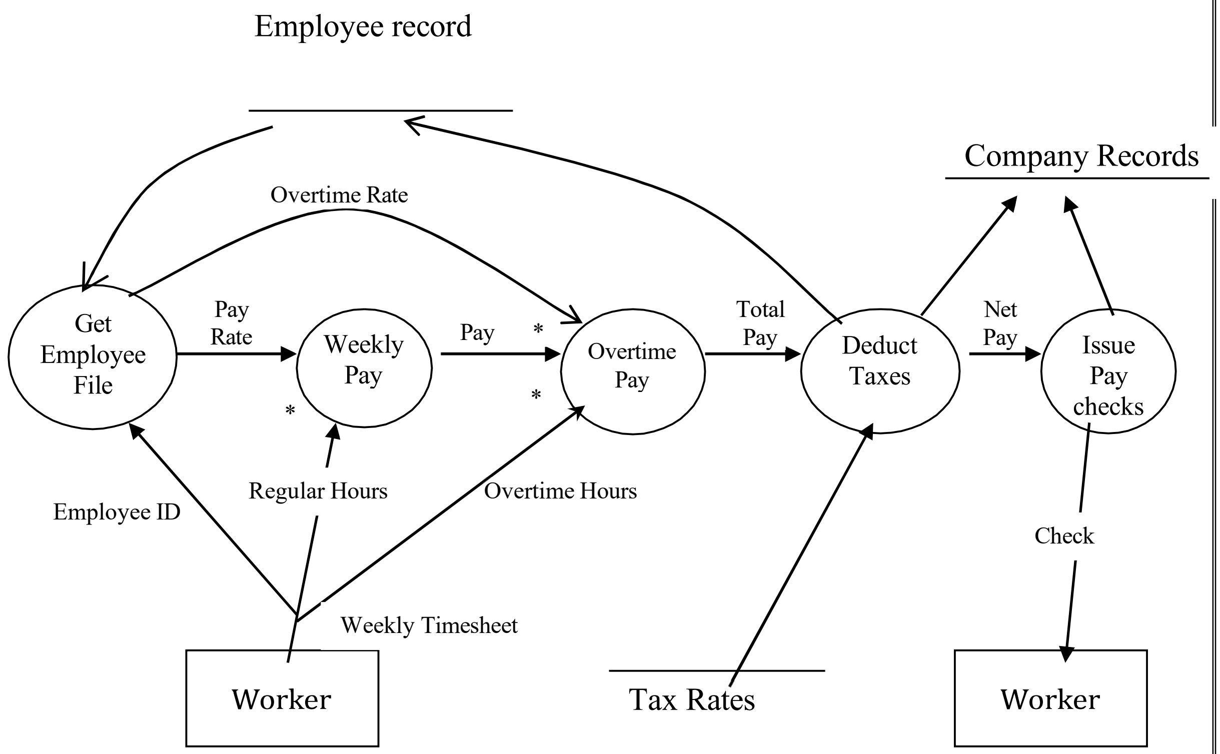


Figure: DFD of a system that pays workers

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:

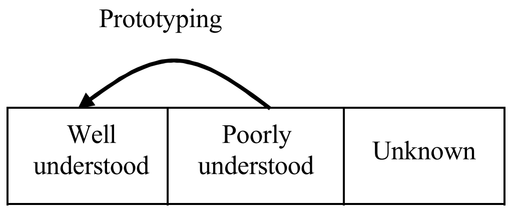
1. Unlabelled Data flows
2. Missing data flows (information required by a process is not available)

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.

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.

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.

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

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.

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.

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|…)*

Structure of an SRS

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
    - Purpose
    - Scope
    - Definitions, Acronyms, and Abbreviations
    - References

  1.2 Overview

2. Overall Description
    - Product Perspective
    - Product Functions

User Characteristics

General Constraints

Assumptions and Dependencies

3. Specific Requirements

External Interface Requirements

User Interfaces

Hardware Interfaces

Software Interfaces

Communication Interfaces

Functional Requirements

Mode 1

Functional Requirement 1.1

.

.

3.2.1.n Functional Requirement 1.n

.

.

Mode m

Functional Requirement m.1

Performance Requirements

Design Constraints

Attributes

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.

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_________the phase of the requirement process.
   A. requirement review
   B. requirement validation
   C. **structured problem analysis**
   D. informal problem analysis

7._____represents the 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 requirements.
A. It specifies the performance constraints on the software system
B. It specifies 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 for SRS.
2. Explain the phases of the requirement process with a diagram.
3. Explain briefly the structured analysis technique.
4. Describe the prototyping technique and its types used for problem analysis.

Application

5. Explain the role of DFD and data dictionary. Also explain different symbols with purposes used in DFD.
6. Explain the characteristics of SRS.
7. Explain various components of an SRS.
8. Write the various factors considered in design constraints imposed on implementation.
9. Explain the general structure of SRS.

MODULE - III

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 ca1led 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.

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).

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.

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 for 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.

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.

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.

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 module 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.

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 a 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

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

14. Modules that take information from their superordinate 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 superordinates.

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 mechanisms for software systems.
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.

Application

5. Describe the lowest and strongest level of cohesion with a suitable example.
6. Define the meaning of coupling. Explain the factors that affect coupling.
7. Write a note on SDM strategy.

MODULE-IV
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.

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.

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.

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.

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.

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.

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.

```
if Cl 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:

```
if C1 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 structure. If such situations do arise, the program should not just "crash" or "core jump"; it should produce some meaningful message and exit gracefully.

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 mainteance. 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.

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.

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.

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. Critical 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**
B. goto statement
C. information hiding
D. multiple exits

10.Program having static structure implies ___________.

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

11.Expand PDL.

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

Skills

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

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

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

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

14. In consistency checker ________ cannot be generated.

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

15. What is the linear organization of statement of program referred as ___________.

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

16.Sequence of Statement executed during execution of the program refers to__________.

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

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

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

18.Use of comments is for________.

A. Verification
B. Static analysis
C. **Internal documentation**
D. Data flow anomalies

19.___is methodically analyzing 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. Explain the concept of structured programming.

Application

5. Explain any four programming style.
6. Explain internal documentation and what are the information it contains.
7. Write a note on verification and code reading.

MODULE-V

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.

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 Illustrates this step.

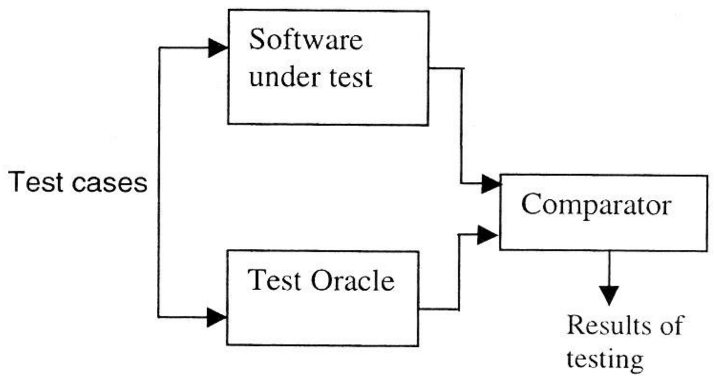


Figure: Test Oracles

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.

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".

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.

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

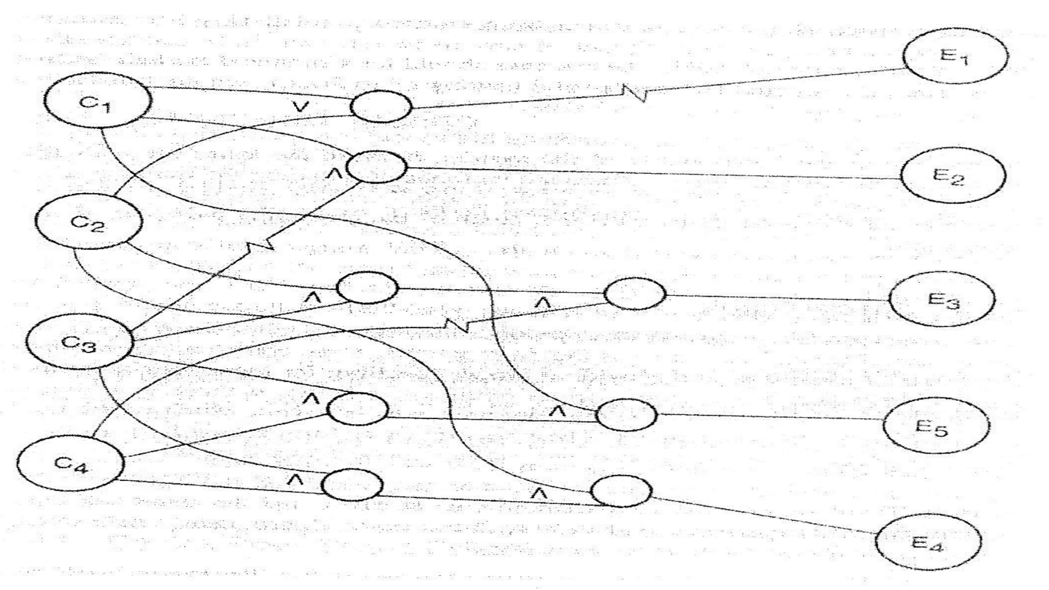


Figure: The Causes effect graph of Structural Testing

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.

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.

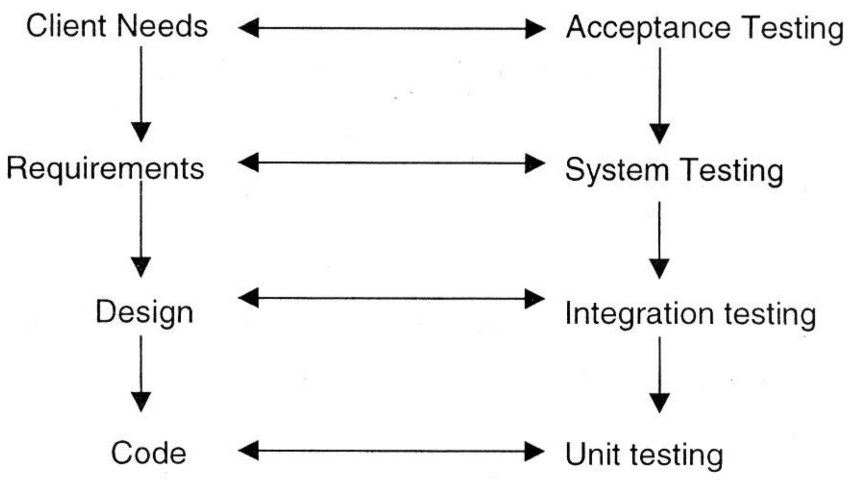


Figure: Levels of testing

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.

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 understanding not only the code but also the related documents. During the modification of the software, the effect of 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. To make the 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 costliest activity in a software product's life.

Maintenance is a set of software engineering activities that occur after the 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 that 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.

PART - A
(Multiple choice questions)
Application

1. _____is a process of analyzing a program to find errors.
   A. Coding
   B. **Testing**
   C. Maintenance
   D. Analyzing

2. _____is a mechanism used to check the correctness output of a program for test cases.
   A. Coding
   B. Testing
   C. Maintenance
   D. **Testing oracles**

3. In the Top-down testing approach what will stimulate the behavior of Sub Routines.
   A. Driver
   B. **Stub**
   C. Causes
   D. test Oracle

4. What are required in a 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 the software is working satisfactorily.

A. Unit testing
B. Integration testing
C. System testing
D. **Acceptance testing**

21. Testing during the maintenance of software is _____________.

A. Unit testing
B. Maintenance testing
C. **Regression testing**
D. Adaptive testing

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

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

23. The activity of adding new capabilities and modifying existing functions is _____________.

A. Corrective maintenance
B. **Perfective maintenance**

C. Adaptive maintenance

D. Preventive maintenance

24. Which maintenance is the largest consumer of maintenance resource.

A. Corrective maintenance

B. **Perfective maintenance**

C. Adaptive maintenance

D. Preventive maintenance

25. The IEEE definition of "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 an example and diagram.
6. Briefly explain structural testing.

SKILLS

7. Explain the control flow-based testing with a suitable example.
8. Explain the data flow-based testing with an example.
9. Explain Mutation Testing.
10. Compare the different levels of testing.
11. What do you mean by maintenance? Explain the software maintenance activities.