SRINIVAS UNIVERSITY

COLLEGE OF COMPUTER SCIENCE & TECHNOLOGY
CITY CAMPUS, PANDESHWAR,
MANGALORE-575 001

Question Answer Bank with Bloom's Model

SOFTWARE ENGINEERING

B.C.A - V SEMESTER

Compiled by
Faculty

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

WEIGHTAGE TO OBJECTIVES TABLE

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

BLUE PRINT

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

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

Unit I
Part A (Multiple Choice Questions)
Remembering

1. IEEE defines _____ as the collection of computer programs,
    procedures, rules and associated documentation and data. Identify the same
    A. SOFTWARE Engineering   C. software model
    B. **software**   D. SRS

2. IEEE defines _____ is a systematic approach to the development, operation,
    maintenance and requirement of the software. Identify the same
    A. **SOFTWARE Engineering**   C. coding
    B. requirement specification   D. SRS

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

    _____
    A. SRS   **D. High quality software**
    B. Review report   **product**
    C. Cost effective design

4. Quote from the memory that _____ is the effort required to couple system with
    other system
    A. Maintainability   C. Portability
    B. **Inter-operability**   D. Quality

5. Quote from memory that _____ is the effort required to transfer the software from
    one hardware configuration to other.
    A. Maintainability   **C. Portability**
    B. Inter-operability   D. Reliability

6. State that _____ is the effort required to locate and fix the errors in the program.
    A. Maintainability   C. Portability
    B. Inter-operability   D. Testing

7. State that _____ is one of the quality attributes in product revision
    A. Portability   C. Integrity
    B. Efficiency   **D. Testability**

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

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

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

Understanding

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

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

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

14.   Development process of software has _____ phases
    A.   Three   C.   Five
    B.   Four   D.   Two

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

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

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

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

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

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

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

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

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

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

25. In SCI life cycle, CR request is approved by_____
    A. **Configuration Manager**
    B. CCB
    C. Project manager
    D. developer

Part B (4 marks)

Remember

1. Define Software Engineering. Explain various problem faced in Software Engineering

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 equipment are made useful to man via computer programs, procedures and associated documentation".

1.2.1. Problem of Scale

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

1.2.2. Cost, Schedule and Quality

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

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

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.

1.2.3. The Problem of Consistency

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

2. Discover the Quality attributes of Software Engineering

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.

3. Discover different phases of 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.

4.   Discover in brief about the characteristics of a software process

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

2.2.1. Predictability

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

Property
of interest
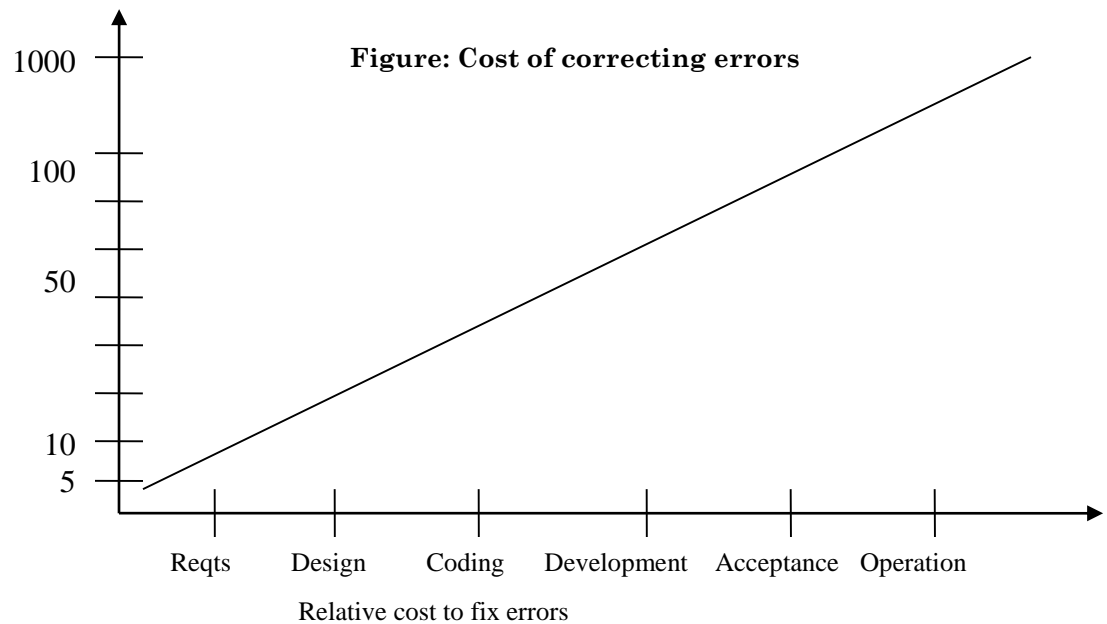
**Process under
statistical control**

Projects
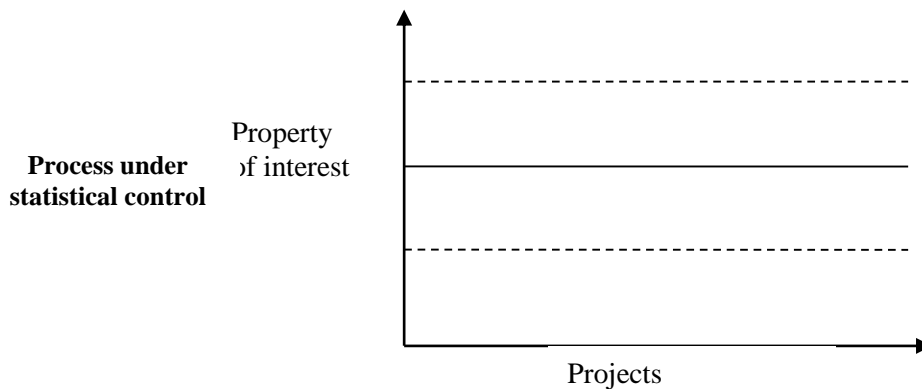
2.2.2. Support Maintainability and Testability

Software products are not only easily maintainable because of the development process which is used for developing the software does not contain maintainability as a clear goal. Developers are made responsible for maintenance at least for a couple of years after developing the software. Many examples show us that, programming is not a major activity where programmer spends his time. Testing consumes most resources during development. The goal of the process should not be to reduce the effort of design and coding but to reduce the effort of testing and maintenance. Both testing and maintenance depend heavily on design and coding and these costs are considerably reduced if the software is designed and coded make testing and maintenance easy.

2.2.3. Early Defect Removal and Defect Prevention

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

Figure: Cost of correcting errors

Relative cost to fix errors

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

5.   State and explain the working of waterfall model with the help of a diagram.

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 .
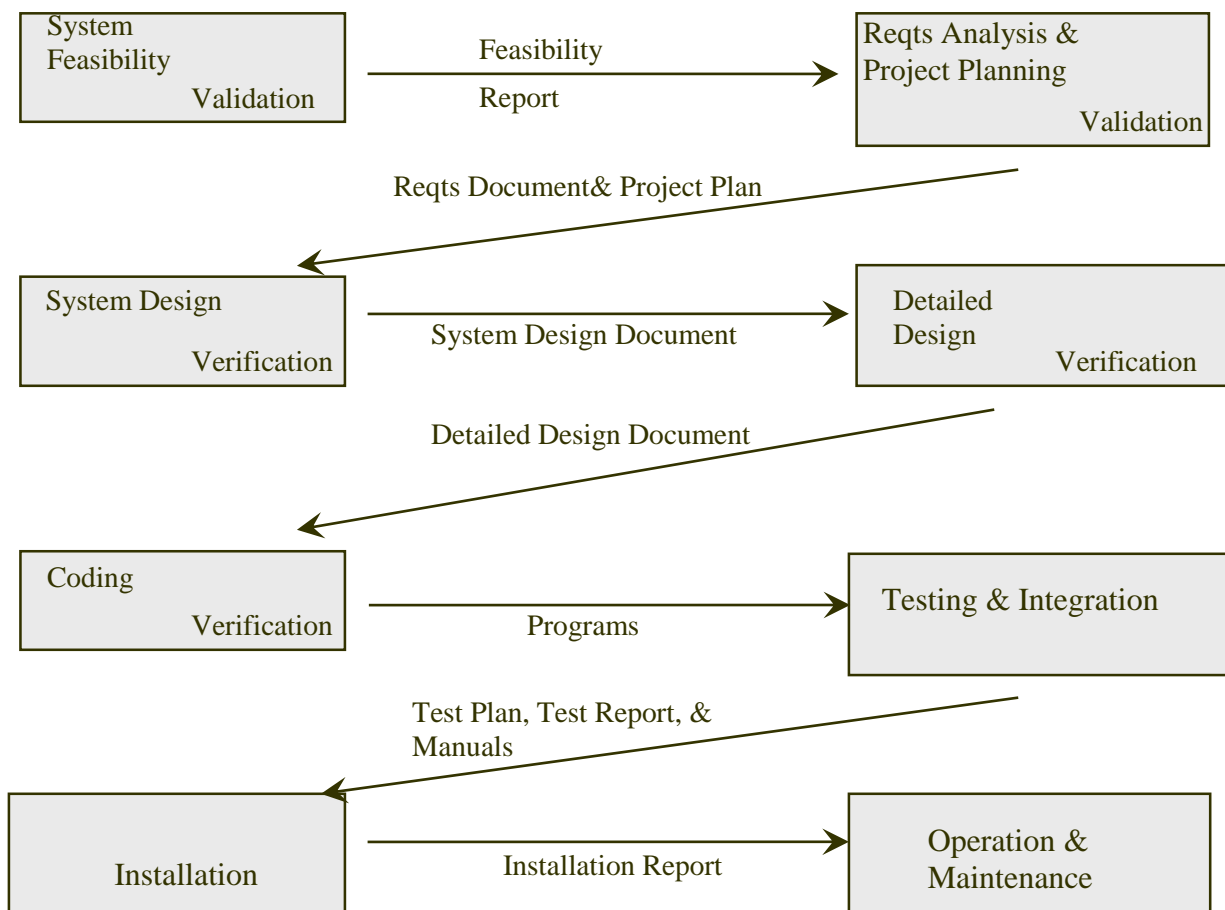
Each phase begins soon after the completion of the previous phase. Verification and validation activities are to be conducted to ensure that the output of a phase is consistent with the overall requirements of the system. At the end of every phase there will be an output. Outputs of earlier phases can 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.

```
┌─────────────────┐                           ┌──────────────────────┐
│ System          │      Feasibility          │ Reqts Analysis &     │
│ Feasibility     │ ─────────────────────────>│ Project Planning     │
│        Validation│     Report                │          Validation  │
└─────────────────┘                           └──────────────────────┘

            Reqts Document& Project Plan

┌─────────────────┐                           ┌──────────────────────┐
│ System Design   │   System Design Document  │ Detailed             │
│                 │ ─────────────────────────>│ Design               │
│     Verification│                           │          Verification│
└─────────────────┘                           └──────────────────────┘

            Detailed Design Document

┌─────────────────┐                           ┌──────────────────────┐
│ Coding          │         Programs          │ Testing & Integration│
│     Verification│ ─────────────────────────>│                      │
└─────────────────┘                           └──────────────────────┘

      Test Plan, Test Report, &
      Manuals

┌─────────────────┐                           ┌──────────────────────┐
│                 │    Installation Report    │ Operation &          │
│ Installation    │ ─────────────────────────>│ Maintenance          │
└─────────────────┘                           └──────────────────────┘
```

Waterfall Model

6.   List out the limitations of 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.

Understand

7.   Explain the working of iterative enhancement model

2.3.4. Iterative Enhancement Model

Design $_0$	Design $_1$	Design $_n$
Implement $_0$	Implement $_1$	Implement $_n$
Analysis $_0$	Analysis $_1$	Analysis $_n$

Iterative Enhancement Model
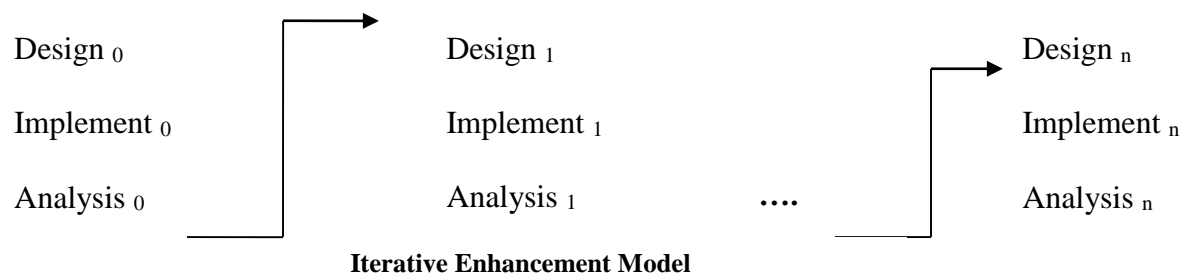
This model tries to combine the benefits of both prototyping and waterfall model. The basic idea is, software should be developed in increments, and each increment adds some functional capability to the system. This process is continued until the full system is implemented. An

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.

8.   Describe the spiral model with the help of diagram

2.3.5. Spiral Model

As the name suggests, the activities of this model can be organized like a spiral that has many cycles as shown in the above figure. Each cycle in the spiral begins with the identification of objectives for that cycle; the different alternatives that are possible for achieving the objectives and the constraints that exist. This is the first quadrant of the cycle. The next step is to evaluate different alternatives based on the objectives and constraints. The focus is based on the risks. Risks reflect the chances that some of the objectives of the project may not be met. Next step is to develop strategies that resolve the uncertainties and risks. This step may involve activities like

prototyping.

Cumulative cost

Progress through steps

Determine objectives, alternatives, constraints

Evaluate alternatives, identify, resolve risks

Risk analysis

Risk analysis

Risk analysis

Risk analysis

Risk analy-sis

Prototype 1

Prototype 2

Prototype 3

Operational prototype

Review

Commitment partition

Simulations, models, benchmarks

Requirements plan life-cycle plan

Concept of operation

Software requirements

Software product design

Detailed design

Develop-ment plan

Requirements validation

Code

Unit test

Integration and test plan

Design validation and verification

Integration and test

Implementation

Acceptance test

Plan next phases

Develop, verify next-level product

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.

9. **Explain the phases involved in project management process**

2.4.1. Phases of Management Process

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

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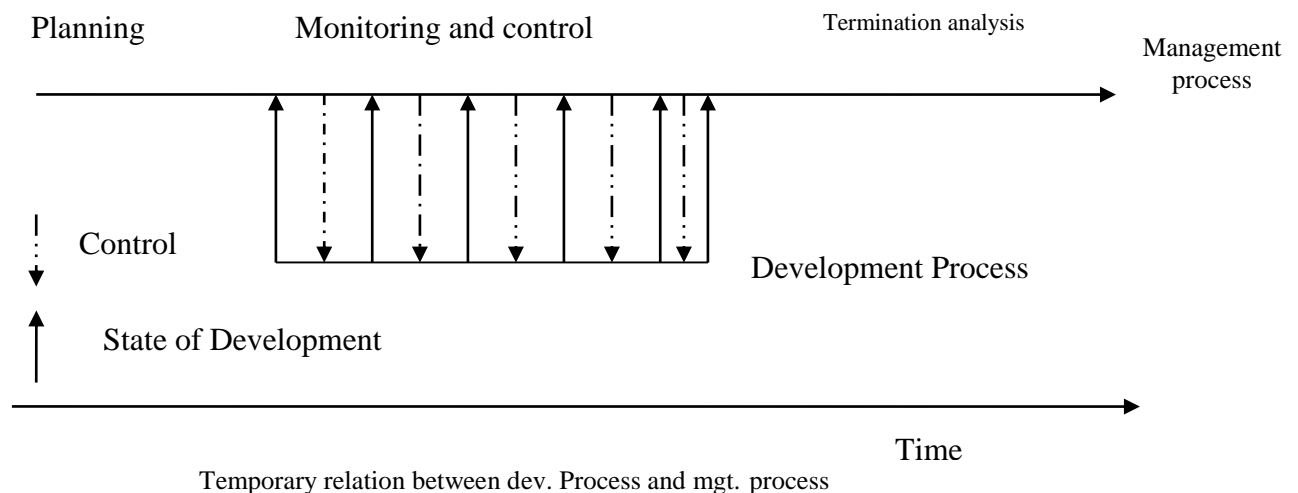
Project monitoring and control phase includes all activities that the project management has to perform while development is going on to ensure that project objectives are met and the development process proceeds according to the plan. If the objectives are not met, then this phase exerts suitable actions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

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

10.   Describe the SCM life cycle of an item

2.5.1. Configuration Identification:

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

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

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

2.5.2. Change Control:

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

When a SCI is under development, it has considered being in working state. It is not under SCM and can be changed freely. Once the developer is satisfied with the SCI, then it is given to CM for review and the item enters to 'under review' state. The CM reviews the SCI and if it is approved, it enters into a library after which the item is formally under SCM. If the item is not approved, the item is given back to the developer. This cycle of a SCI is given in the figure below.
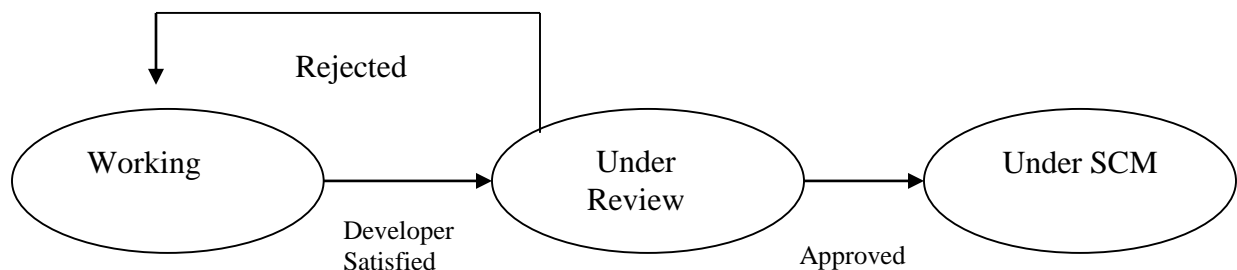
Once the SCI is in the library, it can not be modified, even without the permission of the CM. an SCI under SCM can be changed only if the change has been approved by the CM. A change is initiated by a change request (CR). The reason for change can be anything. The CM evaluates the CR primarily by considering the effect of change on the cost schedule and quality of the project and the benefits likely to come due to this change. Once the CR is accepted, project manager will take over the plan and then CR is implemented by the programmer.

2.5.3. Status Accounting and Auditing

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

11.   Explain various activities of Software configuration Management Process

Same as above

12.   Identify the capability maturity Model

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

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

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

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

Continuously
improving process

Optimizin
g
(5)

Predictable
process

Managed
(4)

Standard,
Consistent
process

Defined
(3)

Disciplined
process

Repeatabl
e
(2)

Initial
(1)
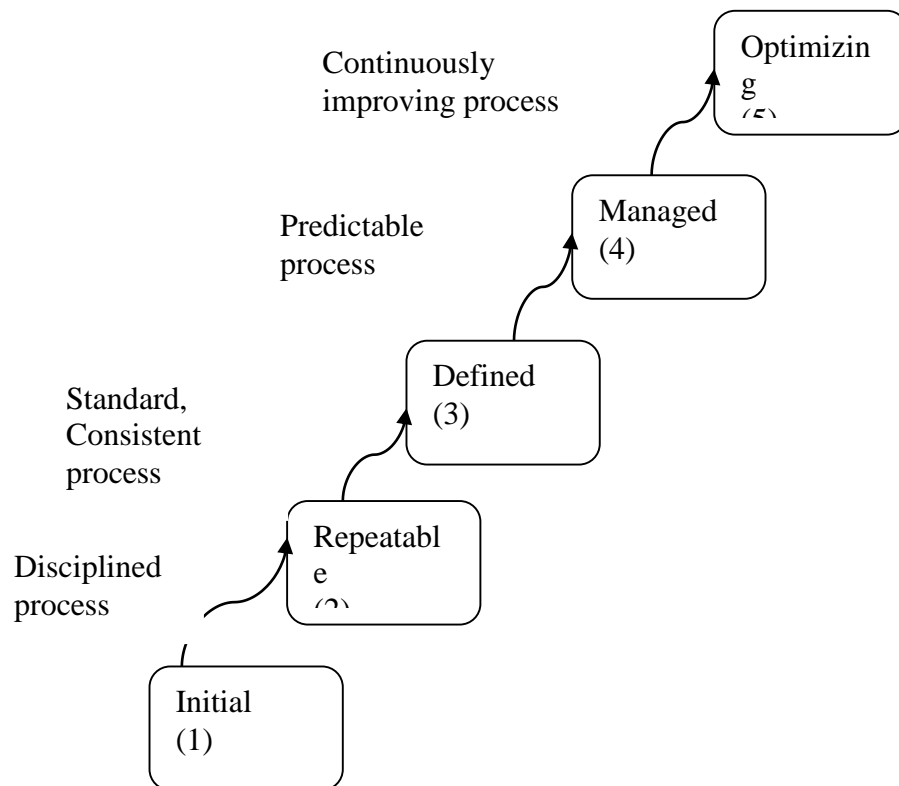
At the managed level (level 4) Data is collected from software processes, which is used to build models to characterize the process.

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

UNIT II
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   C.   five
    B.   three   D.   two

3. The author of SRS document is____
    A.   developer   C.   end user
    B.   client   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   C. the User and the Analyst
    B. the Client and the Designer   D. the Client and the Analyst

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

7. _____ represents Flow of data in a system
    A.   DFD   C.   Finite State Automata
    B.   Flow chart   D.   SRS

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

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

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

11.   ____specifies the repeated occurrence of a regular expression

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

12.   An Evolutionary prototype leads to _____model

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

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

14.   if requirement stated doesn't conflict with another, then SRS is said to be___
        A.   Consistent   C.   unambiguous
        B.   traceable   D.   verifiable

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

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

17.   what is the error type ,if some requirements are not included in SRS
        A.   Inconsistency   C.   Incorrect Fact
        B.   Omission   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   **C.   Prototyping**
      B.   Finite state automata   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   C.   Parallel lines
      B.   Rectangle   D.   Named Arrows

22.   Redundancy is a major issue in_____
      A.   Consistency   C.   traceability
      B.   Completeness   **D.   <u>modifiability</u>**

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

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

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

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

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

Part B (4 marks)
Understanding

1.   **Explain the need of 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.

2.   Explain the phases of requirement process with diagram

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.

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.

Figure (a)
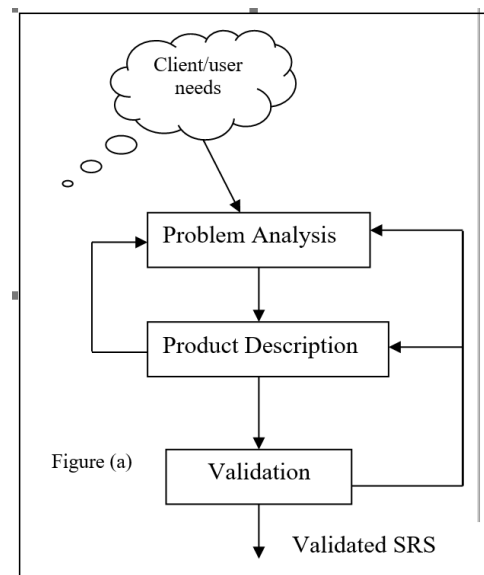
3. Explain the role of analyst in problem Analysis

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

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

The role of the analysts include

- Study of the current system to be automated
- Meeting the various stack holders
- Studying the different sources of information for collecting the requirements
- Collecting requirements
- Validating the requirements

4. Describe the informal approach of problem analysis

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

5. Explain briefly structured analysis technique

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.
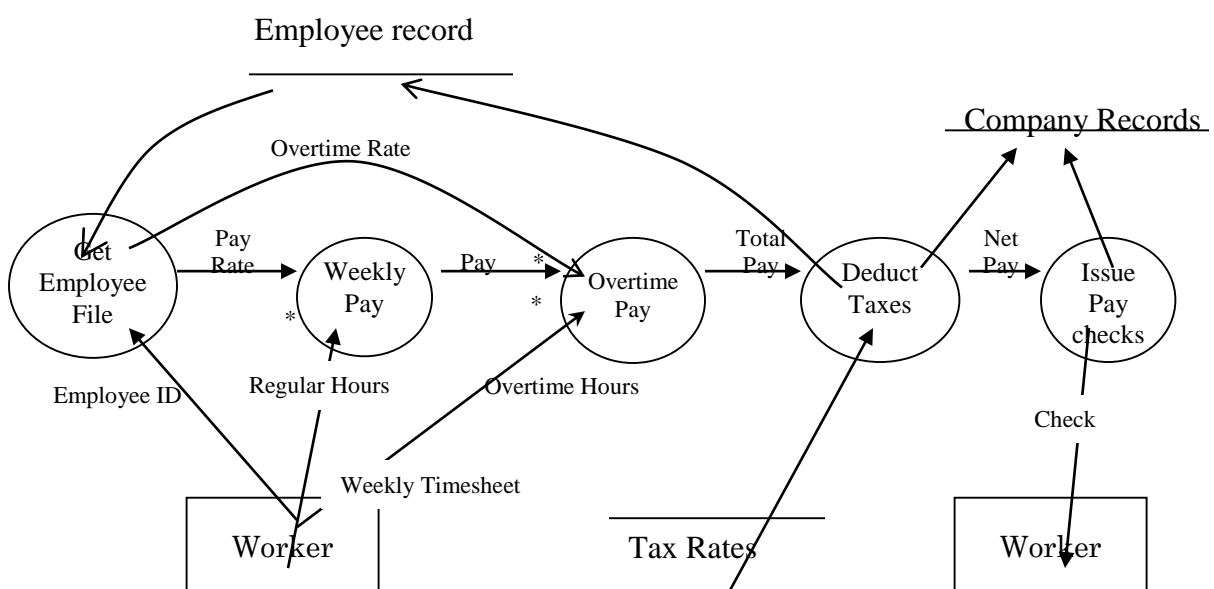
6. **Describe the prototyping technique and its types used for problem analysis**

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.

Prototyping

Well understood	Poorly understood	Unknown

Application
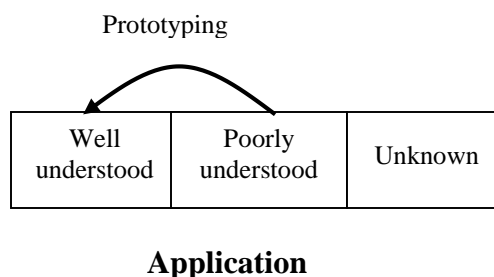
7.   Administer the role of DFD and data dictionary? Write different symbol with purpose used in DFD

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

The symbols used in DFD are

Notation	De Marco & Yourdon	Gane and Sarson
External Entity		
Process		
Data Store		
Data Flow		

8.   Interpret the characteristics of SRS

A SRS is correct 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 consistent 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.

9.   **Interpret various components of an 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.

10.   Write the various factors considered in design constraint imposed on implementation

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.

11. **Write the general structure of SRS, explain in brief**

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

The general structure of an SRS is given below.

1.   Introduction

1.1.1 Purpose

1.1.2   Scope

1.1.3   Definitions, Acronyms, and Abbreviations

1.1.4   References

1.2   Overview

2.   Overall Description

2.1   Product Perspective

2.2   Product Functions

2.3   User Characteristics

2.4   General Constraints

2.5   Assumptions and Dependencies

3.   Specific Requirements

3.1   External Interface Requirements

3.1.1   User Interfaces

3.1.2   Hardware Interfaces

3.1.3   Software Interfaces

3.1.4   Communication Interfaces

3.2   Functional Requirements

3.2.1   Mode 1

3.2.1.1 Functional Requirement 1.1

.

.

3.2.1.n   Functional Requirement 1.n

.

.

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

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

UNIT III
PART A (Multiple Choice Questions)
Application

1.   For a function-oriented design, the design can be represented graphically by _____
   A.   Structure Charts   C.   DFD
   B.   System Chart   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   **C.   Abstraction**
   B.   Cohesion   D.   Factoring

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

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

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

6.   _____ is the lowest level of cohesion
   A.   Coincidental   C.   Temporal
   B.   Logical   D.   Procedural

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

9.   Which cohesion occurs when there is *no* meaningful relationship among the elements of a module.
   A.   Coincidental   C.   Temporal
   B.   Logical   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	C.	**Sequential**
B.	Communicational	D.	Functional

11. Which of the following is not a type of module

A.	Co-ordinate	**C.**	**Logical**
B.	Output	D.	transform

12. Words like "initialize," and "cleanup" imply _____cohesion.

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

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

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

14. *modules* that take information from their superiordinate and pass it on to its subordinates.

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

15. modules whose primary concern is managing the flow of data to and from different subordinates

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

16. The main module is a _____module.

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

17. Which is the major part of system design specification.

A.	**Module specification**	C.	Coupling
B.	First level factoring	D.	Cohesion

18. There are _____ major steps in Structured Design Methodology (SDM)

A.	Seven	C.	Five
B.	Six	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**   C. modularization
    B. *abstraction*   D. coupling

21. _____is considered the indication of module complexity.

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

22. _____of a module is the number of arrows coming towards the module indicating the number of superordinate.

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

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

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

25. What is the aim of the design review.

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

Part-B
Understanding

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

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

the behavior of the entire system can be understood. This also helps determine how modifying a component affects the system. During design, the components do not exist, and in the design the designer specifies only the abstract specifications of the different components. The basic goal of system design is to specify the modules in a system and their abstractions. Once the different modules .are specified, during the detailed design the designer can concentrate on one module at a time. The task in detailed design and implementation is essentially to implement the modules so that the abstract specifications of each module are satisfied.

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

2.   Explain the two modularizing criteria for functional abstraction

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.

3.   Identify the meaning of cohesion. Describe the different levels of 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.

4.   Explain any two types of cohesion.
Any two types f the above must be answered.

5.   Describe the lowest and strongest level of cohesion with suitable example
Explanation of coincidental cohesion and functional cohesion is required

6.   Explain different types of modules in a structure chart.

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

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

Data to Superiordinate   Data from Superiordinate

```
  ┌─────────┐                          ┌─────────┐
  │  Input  │             ↓            │ Output  │
  │ Module  │                          │ Module  │
  └─────────┘                          └─────────┘
```
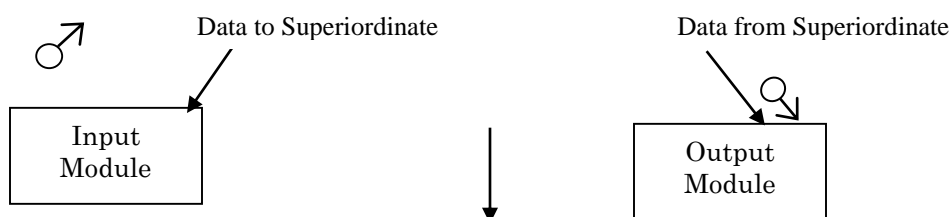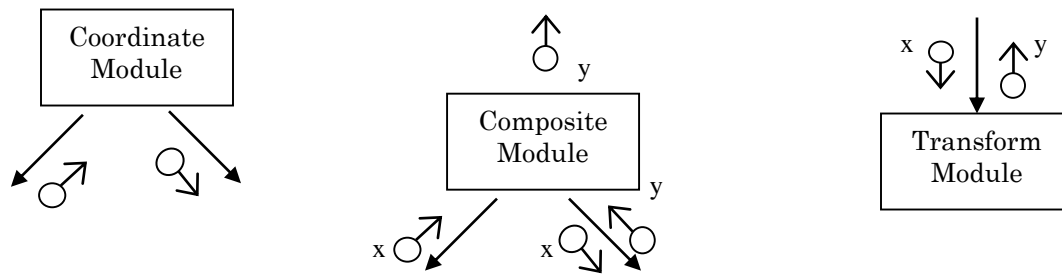
Figure 4.3: Different types of modules

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

7.   Interpret the meaning of coupling? Explain the factors that affect 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.

Application

8. Discover the input, output and transform branch factoring with example

The factoring of the input module get-sorted-list in the first-level structure is shown in Figure 4.6.

The transform producing the input returned by this module (i.e., the sort transform) is treated as a central transform. Its input is the word list. Thus, in the first factoring we have an input module to get the list and a transform module to sort the list. The input module can be factored further, as the module needs to perform two functions, getting a word and then adding it to the list. Note that the looping arrow is used to show the iteration. The factoring of the output modules is symmetrical to the factoring of the input modules. For an output module we look at the next transform to be applied to the output to bring it closer to the ultimate desired output. This now becomes the central transform, and an output module is created for each data stream. Factoring the central transform is essentially an exercise in functional decomposition and will depend on the designers' experience and judgment. One way to factor a transform module is to treat it as a problem in its own right and start with a data flow diagram for it. The factoring of the central transform count-the-number-of-different-words is shown in Figure 4.7.
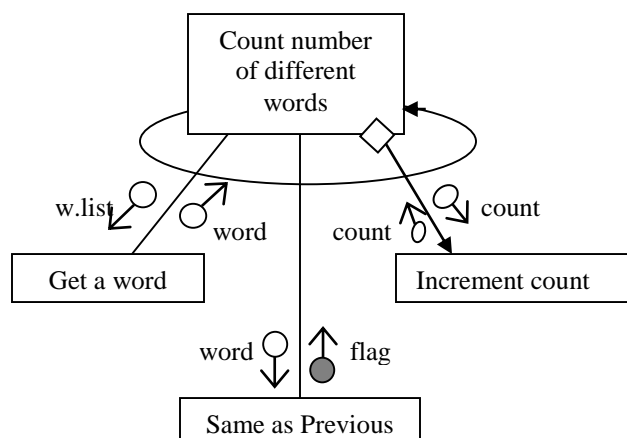
This was a relatively simple transform, and we did not need to draw the data flow diagram. To determine the number of words, we have to get a word repeatedly, determine if it is the same as the previous word (for a sorted list, this checking is sufficient to determine if the word is different from other words), and then count the word if it is different. For each. of the three different functions, we have a subordinate module, and we get the structure shown in Figure 4.7.

9. **Illustrate the various factors considered in design heuristics of detailed design**

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

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

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

Another factor to be considered is "**fan-in" and "fan-out**" of modules. Fan-in of a module is the number of arrows coming towards the module indicating the number of superiordinates. Fan-out of a module is the number of arrows going out of that module; indicating the number of

subordinates for that module. A very-high fan-out is not desirable as it means that the module has to control and co-ordinate too many modules. Whenever possible, fan-in should be maximized. In general, the fan-out should not be more than 6.

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

10.   Develop the differences between fan in and fan out factoring design heuristic

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

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

11.   Write a note on SDM strategy.

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

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

1.   Restate the problem as a data flow diagram
2.   Identify the input and output data elements
3.   First-level factoring
4.   Factoring of input, output, and transform branches

12.   Demonstrate the concept of Factoring? Explain First level factoring with example

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:

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

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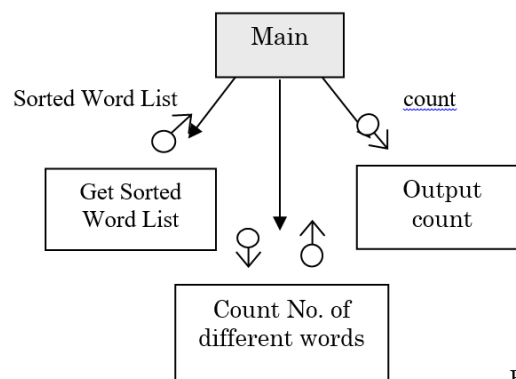

Figure 4.5: First-level factoring

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

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

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

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

The critical design review process is similar to the other reviews, in that a group of people get together to discuss the design with the aim of revealing designs errors or undesirable properties. The review group includes, besides the author of detailed design, a member of the system design team, the programmer responsible for ultimately coding the module(s) under review, and an independent software quality engineer. That is, each member studies the design beforehand and with aid of a checklist marks items that the reviewer feels are incorrect or need c1arification. The members ask questions and the designer tries to explain the situation. During the discussion design errors are revealed. As with any review, it should be kept in mind that the aim of the meeting is to uncover design errors, not try to fix them. Fixing is done later. Also, the psychological frame of mind should be healthy, and the designer should not be put in a defensive position. The meeting

should end with a list of action items, to be acted on later by the designer. The use of checklists, as with other reviews, is considered important for the success of the review.

5.3.3. Consistency Checkers

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

UNIT IV

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   C. **Black box**
      B. Block 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**   C. Post-condition
      B. logical-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   C. **Post-condition**
      B. logical-condition   D. E Prime

4.Which language is a way to communicate a design precisely and completely.
      A. **Process Design Language**   C. Process Definition Language
      B. Program Design Language   D. Process Definition Language

5.Which are not consider as verification method for design phase.
- A. Design walkthrough
- B. 2Critical design review
- C. Consistency checkers
- D. **Symbolic execution**

6.Which is a manual method of verification in detailed design phase.
- A. **Design walkthrough**
- B. Critical design review
- C. Consistency checkers
- D. Path condition

7.____ are compilers which take design specified in PDL as input
- **A. Consistency Checkers**
- B. Execution tree
- C. Drivers
- D. stubs

8.Structured programming is often regarded as_____programming.
- A. **Goto-less**
- B. Goto with
- C. Dynamic
- D. stepwise refinement

9.The key property of a structured Programming is that it has a
- **A. single-entry and a single-exit**
- 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. Process Definition Language
- D. Process Definition Language

Skill

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 conditon
- **B. Critical Design Review**
- C. Algorithm design
- D. Symbolic execution

14. In consistency checker _____ can not 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 analysing the program text.
   A. code reading
   B. **Static analysis**
   C. Internal documentation
   D. code Inspection

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

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

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

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

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

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

PART B
Understanding

1.   Explain module specifications in detailed design.

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

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

An important property of specifications is that they should be implementation independent. Specifications should be given in an abstract manner independent of the eventual implementation of the module and should not specify or suggest any particular method for implementation of the module. This property specifically rules out algorithmic methods for specification. The specification should only give the external behavior; the internal details of the module should be decided later by the programmer.

2.   **Explain PDL with suitable example.**

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.

3.   **Describe the Logic/Algorithm design.**

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

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

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

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

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

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

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

Figure (b): Refinement of the reading operation.

4.   **Identify the three verification method of a detailed design. Explain any one.**

5.3.1 Design Walkthroughs

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

5.3.2 Critical Design Review

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

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

5.3.3. Consistency Checkers

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

5.   Observe and interpret the activities that are undertaken during critical design review.

6.   Discuss on the concepts i. Design walkthroughs ii. Consistency checkers.

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

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

Application

7.   Explain the concept of 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.

8.   Write a note on data flow anomalies with example

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

x = a;

x does not appear in any right hand side

x = b;

FIGURE 8.2. A code segment.

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

9.   Explain any four programming style.

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

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

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

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

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

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

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

ifCl then Sl

else if C2 then S2

else if C3 then S3

else if C4 then S4;

If the different conditions are disjoint (as they often are), this structure can be converted into the following structure:

ifC1 then Sl;

if C2 then S2;

if C3 then S3;

if C4 then S4;

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

10.   Explain internal documentation and what are the information it contains.

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

1. Module functionality, or what the module is doing.

2. Parameters and their purpose.

3. Assumptions about the inputs, if any.

4. Global variables accessed and/or modified in the module.

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

11.   Explain the symbolic execution and execution tree with an example

Here the program is "symbolically executed" with symbolic data. Hence the inputs to the program are not numbers but symbols representing the input data, which can take different values. The execution of the program proceeds like normal execution, except that it deals with values that are not numbers but formulas consisting of the symbolic input values. The outputs are symbolic formulas of input values. These formulas can be checked to see if the program will behave as expected. This approach is called as symbolic execution.

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

Example:

1. function product (x, y, z: integer): integer;

2. var tmpl, tmp2: integer;

3. begin .

4. tmpl := x*y;

5. tmp2 := y*z;

6. product := tmpl *tmp2/y;

7. end

FIGURE 8.3. Function to determine product.

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

The symbolic execution of the function product

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

The trace of the symbolic execution:

1.   function max (x, y, z: integer) : integer
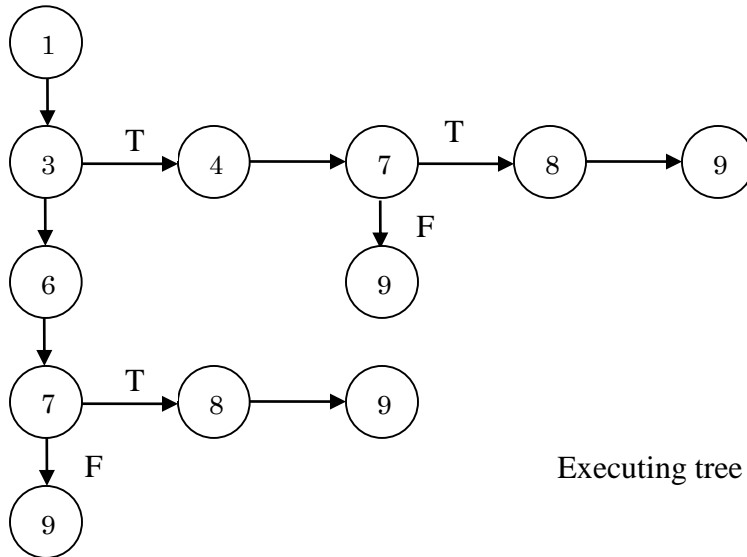
2.   begin

3.   if (x<=y) then

4.   max:=y;

5.   else

6.   max:=x

7.   if max <z then

8.   max:=z;

9.   end;

the code for function max

Executing tree for the function max

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

12.   Explain static analysis and its uses.

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

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

x = a;

x does not appear in any right hand side

x = b;

FIGURE 8.2. A code segment.

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

UNIT V

PART A
MULTIPLE CHOICE QUESTIONS:
Application

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

   A. Coding   C. Maintenance

   B. Testing   D. Analyzing

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

   A. Coding   C. Maintainance

   B. Testing   **D. Testing oracles**

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

   A. Driver   C. Causes

   B. Stub   D. test Oracle

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

   A. test oracle   **C. Drivers**

   B. Cause   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   **C. Errors**

   B. Failure   D. Fault

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

   A. Failure   C. Errors

   B. Testing   D. Fault

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

A. **Test cases**   C. Testing principle

B. Test oracles   D. Testing approach

8.The ___testing procedure is exhaustive testing.

A. Structural testing   C. Unit testing

B. **Functional 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   C. Cause-Effect Graphing

B. **Boundary value analysis**   D. Functional testing

10.____is known as "Glass Box" testing.

A. Functional testing   C. Unit testing

B. **Structural testing**   D. System testing

11. ____is known as "Block Box" testing.

A. **Functional testing**   C. Unit testing

B. Structural testing   D. System testing

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

A. **Path testing**   C. Mutation testing

B. Branch testing   D. Acceptance testing

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

A. Mutation testing   C. Control Flow-based testing

B. **Data flow based testing**   D. Structural testing

14.C-use variable represents

A. Code use of variable   C. Control use of variable

B. **Computational 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   C. DEF variable

    B. P-use 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   C. Control Flow-based testing

    B. Data 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   C. System testing

    B. Integration testing   D. Acceptance testing

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

    A. Unit testing   C. System testing

    B. Integration testing   D. Acceptance testing

19. Where the entire software is tested.

    A. Unit testing   **C. System testing**

    B. Integration testing   D. Acceptance testing

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

    A. Unit testing   C. System testing

    B. Integration testing   **D. Acceptance testing**

21. A Testing during maintainence of software is_____

    A. Unit testing   **C. Regression testing**

    B. Maintainence testing   D. Adaptive testing

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

A. Corrective maintenance   **C. Adaptive maintenance**

B. Perfective maintenance   **D.** Preventive maintenance

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

A. Corrective maintenance   C. Adaptive maintenance

B. Perfective maintenance   **D.** Preventive maintenance

24.Which maintenance is largest consumer of maintenance resource

A. Corrective maintenance   C. Adaptive maintenance

B. Perfective maintenance   D. Preventive maintenance

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

A. Corrective maintenance   C. Adaptive maintenance

B. Perfective maintenance   **D. Preventive maintenance**

PART B
APPLICATION

1.   Explain the test oracle with the help of the diagram.

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

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

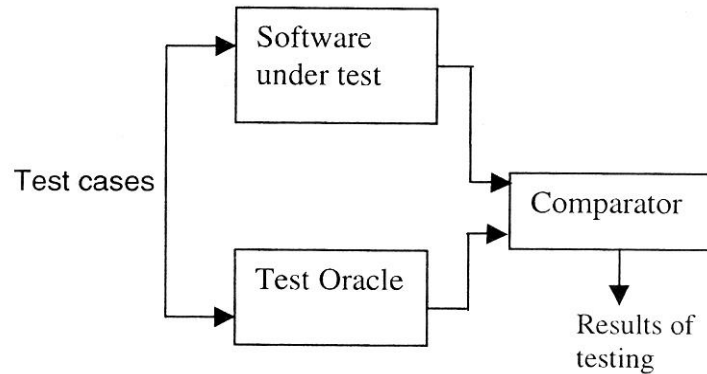Figure 7.1 Illustrates this step.

Figure 7.1 Test Oracles

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

2.   Briefly explain functional testing.

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

Equivalence class partitioning

In this method the domain of all the inputs are divided into a set of equivalence classes so that if any tests in that class succeed, then every test in that class will succeed. That is, we want to identify classes of test such that the success of one test case in a class implies the success of others. However, without looking at the internal structure of the program, it is impossible to determine such ideal equivalence classes. The equivalence class partitioning method tries to approximate this ideal. Putting inputs for which the behavior pattern of the module is specified to be different into similar group's forms different equivalence classes. For example, the specification module that determines the absolute value for integers specifies one behavior for positive integers and another behavior for negative integers. In this case, we will form two equivalence classes-one

consisting of positive integers and the other consisting of negative integers. It is often useful to consider equivalence classes in the output. For an output equivalence class, the goal is to generate test cases such that the output for that test case lies in the output equivalence class. Determining test cases for output classes may be more difficult, but output classes have been found to reveal errors that are not revealed by just considering the input classes.

Boundary value analysis

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

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

Cause-Effect Graphing

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

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

identify the combinations of conditions that make different effects true. A test case is then generated for each combination of conditions, which make some effects true.

3.   Explain 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.

4.   Explain 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.

5. Explain the cause-effect graphing with the an example and diagram.

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

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

Example

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

Credit   acct-number   transaction-amount

Debit   acct-number   transaction-amount

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

Causes:

C1 Command is credit.

C2. Command is debit

C3. Account number is valid.

C4. Transaction amount is valid.

Effects:

E1. Print" invalid command"

E2. Print "invalid account number ".

E3. Print "Debit amount no valid"

E4. Debit account.

E5. Credit account.

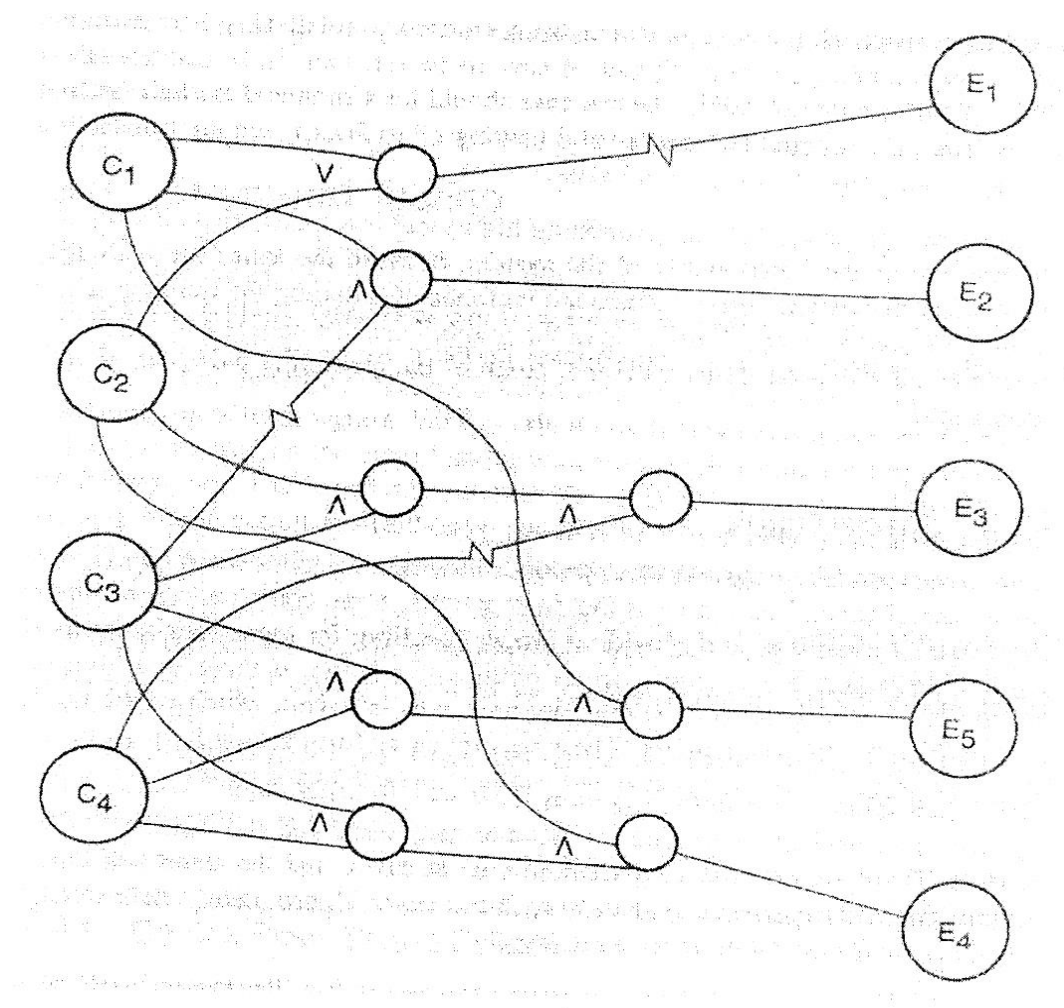The graph representation of Causes-Effect has shown in figure 6.2

Figure 6.2 the Causes effect graph

6.   Briefly explain 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.

SKILL DEVELOPMENT

7.   Justify the control flow based testing with suitable example.

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.

8.   Justify the data flow based testing with an example.

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.

9.   Express the view on 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.

10.   Compare the different levels of testing.

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

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

The relation of the faults introduced in different phases, and the different levels of testing are shown in figure 6.3.
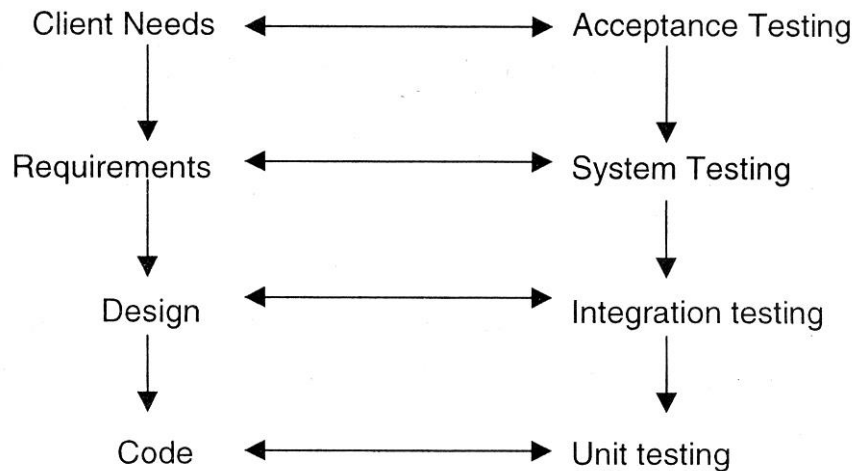
```
Client Needs  ◄──────────►  Acceptance Testing
      │                              │
      ▼                              ▼
Requirements  ◄──────────►  System Testing
      │                              │
      ▼                              ▼
   Design     ◄──────────►  Integration testing
      │                              │
      ▼                              ▼
    Code      ◄──────────►  Unit testing
```

Figure 6.3 Levels of testing.

Unit testing

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

Integration testing

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

System testing

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

Acceptance testing.

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

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

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

11. Focus on test plan.

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

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

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

One of the most important activities of the test plan is to identify the test units A test unit is a set of one or more modules, together with associated data, that are from single computer program and

that are object of testing. A test unit can occur at any level and can contain from a single module to the entire system.

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

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

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

12.   Summarise on maintenance? explain the software maintenance activities

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

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

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

Maintenance activities can be divided into two types:

1.	Modification-As the specifications of computer systems change, reflecting changes in the external world, so must the systems themselves.

2.	Debugging-Removal of errors that should never have been there in the first place.

Software Maintenance Activities

Maintenance can be defined as four activities:

1.	Corrective Maintenance-A process that includes diagnosis and corrective of errors.

2.	Adaptive Maintenance- Activity that modifies software to properly interface with a changing environment (hardware and software).

3.	Perfective Maintenance- Activity for adding new capabilities, modifying existing functions and making general enhancements.

4.	Preventive Maintenance- Activity which changes software to improve future maintainability or reliability or to provide a better basis for future enhancements.

Distribution of maintenance activities

•	Perfective:50%

•	Adaptive:25%

•	Corrective:21%

•	Others(including Preventive):4%