alQuestion Bank
Unit – I
Part – A

1. Define process.

Process, in the software engineering domain, is the set of methods, practices, standards, documents, activities, policies, and procedures that software engineers use to develop and maintain a software system and its associated artifacts, such as project and test plans, design documents, code, and manuals.

2. Differentiate verification and validation.

Validation is the process of evaluating a software system or component during, or at the end of, the development cycle in order to determine whether it satisfies specified requirements Validation is usually associated with traditional execution-based testing, that is, exercising the code with test cases.

Verification is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

3. Define testing and debugging.

Testing is generally described as a group of procedures carried out to evaluate some aspect of a piece of software. Testing can be described as a process used for revealing defects in software, and for establishing that the software has attained a specified degree of quality with respect to selected attributes.

Debugging, or fault localization is the process of (1) locating the fault or defect, (2) repairing the code, and (3) retesting the code.

4. List the benefits of test process improvement.

- smarter testers
- higher quality software
- the ability to meet budget and scheduling goals
- improved planning
- the ability to meet quantifiable testing goals

5. Define Maturity goals.

The maturity goals identify testing improvement goals that must be addressed in order to achieve maturity at that level. To be placed at a level, an organization must satisfy the maturity goals at that level.

6. Define Activities, tasks and responsibilities (ATR).

The ATRs address implementation and organizational adaptation issues at each TMM level. Supporting activities and tasks are identified, and responsibilities are assigned to appropriate groups.

7. List the three critical views (CV) .

Definition of their roles is essential in developing a maturity framework. The manager's view involves commitment and ability to perform activities and tasks related to improving testing capability. The developer/tester's view encompasses the technical activities and tasks that, when applied, constitute quality testing practices. The user's or client's view is defined as a cooperating, or supporting, view.

8. What are the levels of TMM?

Level 1: Initial
Level 2: Phase Definition
Level 3: Integration
Level 4: Management and Measurement
Level 5: Optimization/Defect Prevention and Quality Control

9. Define Errors, Faults (Defects) and Failures.

An error is a mistake, misconception, or misunderstanding on the part of a software developer.

A fault (defect) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification.

Faults or defects are sometimes called "bugs."

A failure is the inability of a software system or component to perform its required functions within specified performance requirements

10. How will the fault manifest itself as a failure?

1. The input to the software must cause the faulty statement to be executed.

2. The faulty statement must produce a different result than the correct statement. This event produces an incorrect internal state for the software.

3. The incorrect internal state must propagate to the output, so that the result of the fault is observable.

11. What are the contents of a test case?

A test case in a practical sense is a test-related item which contains the following information:

1. A set of test inputs. These are data items received from an external source by the code under test. The external source can be hardware, software, or human.

2. Execution conditions. These are conditions required for running the test, for example, a certain state of a database, or a configuration of a hardware device.

3. Expected outputs. These are the specified results to be produced by the code under test.

12. Define Test, Test Oracle and Test Bed.

A test is a group of related test cases, or a group of related test cases and test procedures (steps needed to carry out a test,

A test oracle is a document, or piece of software that allows testers to determine whether a test has been passed or failed.

A test bed is an environment that contains all the hardware and software needed to test a software component or a software system.

13. Define Software Quality.

Two concise definitions for quality are found in the IEEE Standard Glossary of Software Engineering Terminology:

1. Quality relates to the degree to which a system, system component, or process meets specified requirements.

2. Quality relates to the degree to which a system, system component, or process meets customer or user needs, or expectations.

14. Define metric and quality metric.

A metric is a quantitative measure of the degree to which a system, system component, or process possesses a given attribute

A quality metric is a quantitative measurement of the degree to which an item possesses a given quality attribute

15. What are the Quality Attributes?

- Correctness
- reliability
- usability
- integrity
- portability
- maintainability
- interoperability

16. Define Software Quality Assurance Group.

The software quality assurance (SQA) group is a team of people with the necessary training and skills to ensure that all necessary actions are taken during the development process so hat the resulting software conforms to established technical requirements.

17. What are Defect sources?

- Lack of Education
- Poor communication
- Oversight

- Transcription
- Immature process

18. What are the Design Defect Classes?

- Algorithmic and processing
- Control, logic, and sequence
- Data
- Module interface description
- External interface description

19. What are the Coding Defect Classes?

- Algorithmic and processing
- Control, logic, and sequence
- Typographical data flow
- Data
- Module interface
- Code documentation
- External hardware,
- Software

20. What is Defect Repository?

- Defect classes
- Severity
- Occurrences

21. What is a test case?

A test case is a document, which has a set of test data, preconditions, expected results and post conditions, developed for a particular test scenario in order to verify compliance against a specific requirement.

Test Case acts as the starting point for the test execution, and after applying a set of input values, the application has a definitive outcome and leaves the system at some end point or also known as execution post condition.

22. What is the basic objective of software testing?

Software Testing has different goals and objectives. The major objectives of Software testing are as follows:

- Finding defects which may get created by the programmer while developing the software.
- Gaining confidence in and providing information about the level of quality.
- To prevent defects.
- To make sure that the end result meets the business and user requirements.
- To ensure that it satisfies the BRS that is Business Requirement Specification and SRS that is System Requirement Specifications.
- To gain the confidence of the customers by providing them a quality product.

23. List the classification of defects.

Software Defects/ Bugs are normally classified as per:

- Severity / Impact
- Probability / Visibility
- Priority / Urgency
- Related Dimension of Quality
- Related Module / Component
- Phase Detected
- Phase Injected

24. What is design defect?

A design defect is a problem with the product's design that makes the product inherently dangerous or useless, even if it is manufactured perfectly and made of the best-quality materials.

25. What are the V&V task in testing phase?

- Execution of systems test case
- Execution of acceptance test case
- Updating of traceability metrics
- Risk analysis

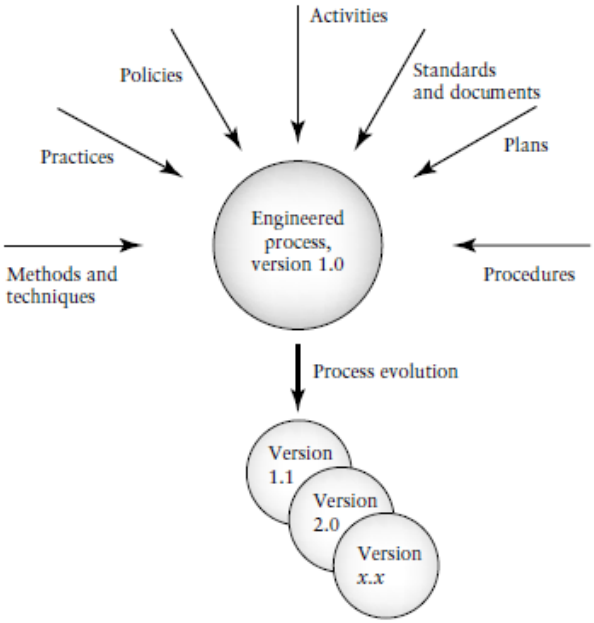
Part – B

1. Explain the role of process in software quality.

The need for software products of high quality has pressured those in the profession to identify and quantify quality factors such as usability, testability, maintainability, and reliability, and to identify engineering practices that support the production of quality products having these favorable attributes.

Among the practices identified that contribute to the development of high-quality software are project planning, requirements management, development of formal specifications, structured design with use of information hiding and encapsulation, design and code reuse, inspections and reviews, product and process measures, education and training of software professionals, development and application of CASE tools, use of effective testing techniques, and integration of testing activities into the entire life cycle.

In addition to identifying these individual best technical and managerial practices, software researchers realized that it was important to integrate them within the context of a high-quality software development process.



It also was clear that adding individual practices to an existing software development process in an ad hoc way was not satisfactory.

The software development process, like most engineering artifacts, must be engineered. That is, it must be designed, implemented, evaluated, and maintained.

As in other engineering disciplines, a software development process must evolve in a consistent and predictable manner, and the best technical and managerial practices must be integrated in a systematic way.

Models such as the Capability Maturity Model_ (CMM)* and SPICE were developed to address process issues.

All the software process improvement models that have had wide acceptance in industry are high-level models, in the sense that they focus on the software process as a whole and do not offer adequate support to evaluate and improve specific software development sub processes such as design and testing.

Most software engineers would agree that testing is a vital component of a quality software process, and is one of the most challenging and costly activities carried out during software development and maintenance.

2. What are the difficulties and challenges for the tester?

Difficulties and challenges for the tester include the following:

- A tester needs to have comprehensive knowledge of the software engineering discipline.
- A tester needs to have knowledge from both experience and education as to how software is specified, designed, and developed.
- A tester needs to be able to manage many details.
- A tester needs to have knowledge of fault types and where faults of a certain type might occur in code constructs.
- A tester needs to reason like a scientist and propose hypotheses that relate to presence of specific types of defects.
- A tester needs to have a good grasp of the problem domain of the software that he/she is testing. Familiarly with a domain may come from educational, training, and work-related experiences.
- A tester needs to create and document test cases. To design the test cases the tester must select inputs often from a very wide domain.
- Those selected should have the highest probability of revealing a defect. Familiarly with the domain is essential.
- A tester needs to design and record test procedures for running the tests.
- A tester needs to plan for testing and allocate the proper resources.
- A tester needs to execute the tests and is responsible for recording results.
- A tester needs to analyze test results and decide on success or failure for a test. This involves understanding and keeping track of an enormous amount of detailed information. A tester may also be required to collect and analyze test-related measurements.
- A tester needs to learn to use tools and keep abreast of the newest test tool advances.
- A tester needs to work and cooperate with requirements engineers, designers, and developers, and often must establish a working relationship with clients and users.
- A tester needs to be educated and trained in this specialized area and often will be required to update his/her knowledge on a regular basis due to changing technologies.

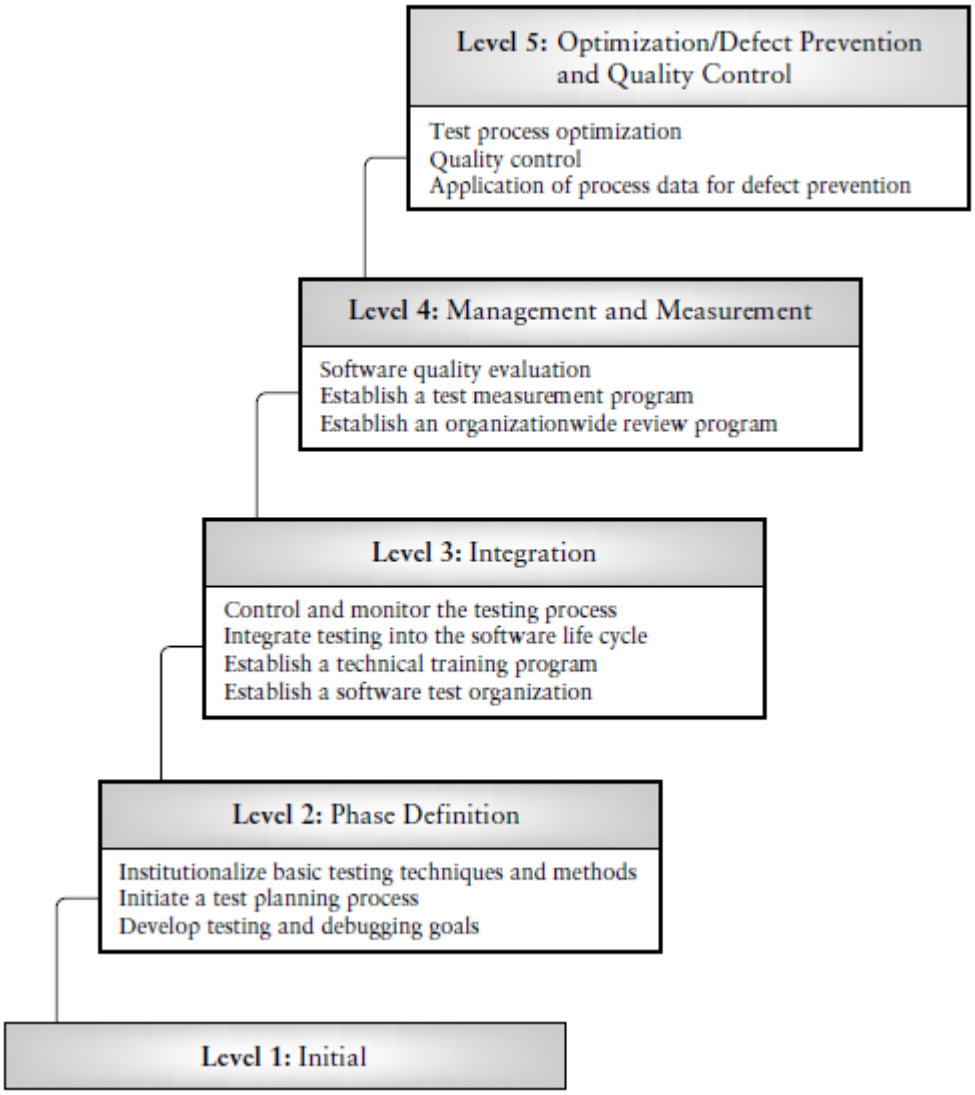
3. Explain 5 level structure of testing maturity model in detail.

The internal structure of the TMM is rich in testing practices that can be learned and applied in a systematic way to support a quality testing process that improves in incremental steps.

There are five levels in the TMM that prescribe a maturity hierarchy and an evolutionary path to test process improvement.

Each level with the exception of level 1 has a structure that consists of the following:

- *A set of maturity goals*. The maturity goals identify testing improvement goals that must be addressed in order to achieve maturity at that level. To be placed at a level, an organization must satisfy the maturity goals at that level. The TMM levels and associated maturity goals.
- *Supporting maturity subgoals.* They define the scope, boundaries and needed accomplishments for a particular level.
- *Activities, tasks and responsibilities (ATR)*. The ATRs address implementation and organizational adaptation issues at each TMM level. Supporting activities and tasks are identified, and responsibilities are assigned to appropriate groups.



Level 1—Initial: (No maturity goals)
At TMM level 1, testing is a chaotic process; it is ill-defined, and not distinguished from debugging. A documented set of specifications for software behavior often does not exist. Tests are developed in an ad hoc way after coding is completed. Testing and debugging are interleaved to get the bugs out of the software

Level 2—Phase Definition:
Goal 1: Develop testing and debugging goals;
Goal 2: Initiate a testing planning process;
Goal 3: Institutionalize basic testing techniques and methods
At level 2 of the TMM testing is separated from debugging and is defined as a phase that follows coding. It is a planned activity; however, test planning at level 2 may occur after coding for reasons related to the immaturity of the testing process. For example, there may be the perception at level 2, that all testing is execution based and dependent on the code; therefore, it should be planned only when the code is complete.

Level 3—Integration:

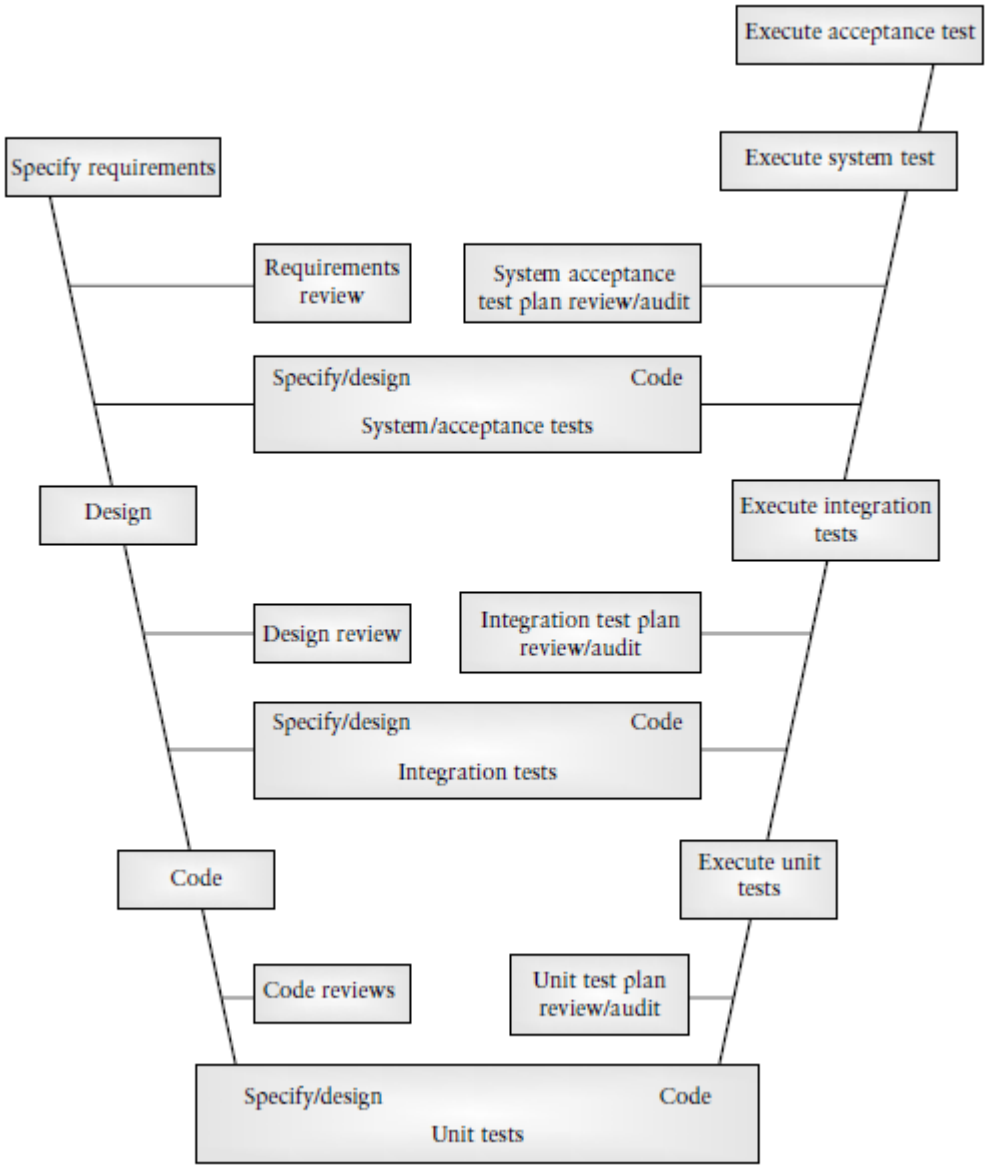
Goal 1: Establish a software test organization;
Goal 2: Establish a technical training program;
Goal 3: Integrate testing into the software life cycle;
Goal 4: Control and monitor testing
At TMM level 3, testing is no longer a phase that follows coding, but is integrated into the entire software life cycle. Organizations can build on the test planning skills they have acquired at level 2. Unlike level 2, planning for testing at TMM level 3 begins at the requirements phase and continues throughout the life cycle supported by a version of the V-model.

Level 4—Management and Measurement:
Goal 1: Establish an organization wide review program;
Goal 2: Establish a test measurement program;
Goal 3: Software quality evaluation
Testing at level 4 becomes a process that is measured and quantified. Reviews at all phases of the development process are now recognized as testing/quality control activities. They are a compliment to execution based tests to detect defects and to evaluate and improve software quality.

Level 5—Optimization/Defect Prevention/Quality Control:
Goal 1:Defect prevention;
Goal 2: Quality control;
Goal 3: Test process optimization
Because of the infrastructure that is in place through achievement of the maturity goals at levels 1–4 of the TMM, the testing process is now said to be defined and managed; its cost and effectiveness can be monitored. At level 5, mechanisms are in place so that testing can be fine-tuned and continuously improved. Defect prevention and quality control are practiced.
Statistical sampling, measurements of confidence levels, trustworthiness, and reliability drive the testing process. Automated tools totally support the running and rerunning of test cases.

4. Describe the test related activities using V model in requirement specification, design, coding and installation phases.

V- model means Verification and Validation model. Just like the waterfall model, the V-Shaped life cycle is a sequential path of execution of processes. Each phase must be completed before the next phase begins. Testing of the product is planned in parallel with a corresponding phase of development in **V-model**.
The various phases of the V-model are as follows:
Requirements like BRS and SRS begin the life cycle model just like the waterfall model. But, in this model before development is started, a system test plan is created. The test plan focuses on meeting the functionality specified in the requirements gathering.
The high-level design (HLD) phase focuses on system architecture and design. It provide overview of solution, platform, system, product and service/process. An integration test plan is created in this phase as well in order to test the pieces of the software systems ability to work together.
The low-level design (LLD) phase is where the actual software components are designed. It defines the actual logic for each and every component of the system. Class diagram with all the methods and relation between classes comes under LLD. Component tests are created in this phase as well.
The implementation phase is, again, where all coding takes place. Once coding is complete, the path of execution continues up the right side of the V where the test plans developed earlier are now put to use.
Coding: This is at the bottom of the V-Shape model. Module design is converted into code by developers.



Validation Phases

Following are the Validation phases in V-Model:

- **Unit Testing:** Unit tests designed in the module design phase are executed on the code during this validation phase. Unit testing is the testing at code level and helps eliminate bugs at an early stage, though all defects cannot be uncovered by unit testing.
- **Integration Testing:** Integration testing is associated with the architectural design phase. Integration tests are performed to test the coexistence and communication of the internal modules within the system.
- **System Testing:** System testing is directly associated with the System design phase. System tests check the entire system functionality and the communication of the system under development with external systems. Most of the software and hardware compatibility issues can be uncovered during system test execution.
- **Acceptance Testing:** Acceptance testing is associated with the business requirement analysis phase and involves testing the product in user environment. Acceptance tests uncover the

compatibility issues with the other systems available in the user environment. It also discovers the non functional issues such as load and performance defects in the actual user environment.

Advantages of V-model:

- Simple and easy to use.
- Testing activities like planning, test designing happens well before coding. This saves a lot of time. Hence higher chance of success over the waterfall model.
- Proactive defect tracking – that is defects are found at early stage.
- Avoids the downward flow of the defects.
- Works well for small projects where requirements are easily understood.

Disadvantages of V-model:

- Very rigid and least flexible.
- Software is developed during the implementation phase, so no early prototypes of the software are produced.
- If any changes happen in midway, then the test documents along with requirement documents has to be updated.

5. Explain software testing principles in detail.

Principles play an important role in all engineering disciplines and are usually introduced as part of an educational background in each branch of engineering.

A principle can be defined as:
1. a general or fundamental, law, doctrine, or assumption;
2. a rule or code of conduct;
3. the laws or facts of nature underlying the working of an artificial device.

Principle 1. Testing is the process of exercising a software component using a selected set of test cases, with the intent of (i) revealing defects, and (ii) evaluating quality.

Software engineers have made great progress in developing methods to prevent and eliminate defects. However, defects do occur, and they have a negative impact on software quality. Testers need to detect these defects before the software becomes operational.

This principle supports testing as an execution-based activity to detect defects. It also supports the separation of testing from debugging since the intent of the latter is to locate defects and repair the software.

The term "software component" is used in this context to represent any unit of software ranging in size and complexity from an individual procedure or method, to an entire software system.

The term "defects" as used in this and in subsequent principles represents any deviations in the software that have a negative impact on its functionality, performance, reliability, security, and/or any other of its specified quality attributes.

Principle 2. When the test objective is to detect defects, then a good test case is one that has a high probability of revealing a yet undetected defect(s).

Principle 2 supports careful test design and provides a criterion with which to evaluate test case design and the effectiveness of the testing effort when the objective is to detect defects.

It requires the tester to consider the goal for each test case, that is, which specific type of defect is to be detected by the test case.

In this way the tester approaches testing in the same way a scientist approaches an experiment. In the case of the scientist there is a hypothesis involved that he/she wants to prove or disprove by means of the experiment.

In the case of the tester, the hypothesis is related to the suspected occurrence of specific types of defects.
The goal for the test is to prove/disprove the hypothesis, that is, determine if the specific defect is present/absent.
Based on the hypothesis, test inputs are selected, correct outputs are determined, and the test is run. Results are analyzed to prove/disprove the hypothesis.

Principle 3. Test results should be inspected meticulously.

Testers need to carefully inspect and interpret test results. Several erroneous and costly scenarios may occur if care is not taken.
For example:A failure may be overlooked, and the test may be granted a "pass" status when in reality the software has failed the test. Testing may continue based on erroneous test results. The defect may be revealed at some later stage of testing, but in that case it may be more costly and difficult to locate and repair.

- A failure may be suspected when in reality none exists. In this case the test may be granted a "fail" status. Much time and effort may be spent on trying to find the defect that does not exist. A careful reexamination of the test results could finally indicate that no failure has occurred.
- The outcome of a quality test may be misunderstood, resulting in unnecessary rework, or oversight of a critical problem.

Principle 4. A test case must contain the expected output or result.

It is often obvious to the novice tester that test inputs must be part of a test case. However, the test case is of no value unless there is an explicit statement of the expected outputs or results,
for example, a specific variable value must be observed or a certain panel button that must light up. Expected outputs allow the tester to determine

(i)whether a defect has been revealed, and
(ii) pass/fail status for the test.

It is very important to have a correct statement of the output so that needless time is not spent due to misconceptions about the outcome of a test. The specification of test inputs and outputs should be part of test design activities.

Principle 5. Test cases should be developed for both valid and invalid input conditions.
A tester must not assume that the software under test will always be provided with valid inputs. Inputs may be incorrect for several reasons. For example, software users may have misunderstandings, or lack information about the nature of the inputs.
They often make typographical errors even when complete/correct information is available. Devices may also provide invalid inputs due to erroneous conditions and malfunctions.
Use of test cases that are based on invalid inputs is very useful for revealing defects since they may exercise the code in unexpected ways and identify unexpected software behavior.
Invalid inputs also help developers and testers evaluate the robustness of the software, that is, its ability to recover when unexpected events occur (in this case an erroneous input)

Principle 6. The probability of the existence of additional defects in a software component is proportional to the number of defects already detected in that component.

What this principle says is that the higher the number of defects already detected in a component, the more likely it is to have additional defects when it undergoes further testing.

For example, if there are two components A and B, and testers have found 20 defects in A and 3 defects in B, then the probability of the existence of additional defects in A is higher than B. This empirical observation may be due to several causes. Defects often occur in clusters and often in code that has a high degree of complexity and is poorly designed.
In the case of such components developers and testers need to decide whether to disregard the current version of the component and work on a redesign, or plan to expend additional testing resources on this component to insure it meets its requirements.
This issue is especially important for components that implement mission or safety critical functions.

Principle 7. Testing should be carried out by a group that is independent of the development group.

This principle holds true for psychological as well as practical reasons. It is difficult for a developer to admit or conceive that software he/she has created and developed can be faulty.
Testers must realize that
(i) developers have a great deal of pride in their work, and
(ii) on a practical level it may be difficult for them to conceptualize where defects could be found.

Even when tests fail, developers often have difficulty in locating the defects since their mental model of the code may overshadow their view of code as it exists in actuality. They may also have misconceptions or misunderstandings concerning the requirements and specifications relating to the software.

Principle 8. Tests must be repeatable and reusable.
Principle 8 calls for experiments in the testing domain to require recording of the exact conditions of the test, any special events that occurred, equipment used, and a careful accounting of the results.
This information is invaluable to the developers when the code is returned for debugging so that they can duplicate test conditions. It is also useful for tests that need to be repeated after defect repair.
The repetition and reuse of tests is also necessary during regression test

Principle 9. Testing should be planned.
Test plans should be developed for each level of testing, and objectives for each level should be described in the associated plan.
The objectives should be stated as quantitatively as possible. Plans, with their precisely specified objectives, are necessary to ensure that adequate time and resources are allocated for testing tasks, and that testing can be monitored and managed.

Principle 10. Testing activities should be integrated into the software life cycle.
It is no longer feasible to postpone testing activities until after the code has been written.
Test planning activities as supported by Principle 10, should be integrated into the software life cycle starting as early as in the requirements analysis phase, and continue on throughout the software life cycle in parallel with development activities.

Principle 11. Testing is a creative and challenging task

6. Explain the tester's role in a software development organization.

- Testing is sometimes erroneously viewed as a destructive activity. The tester's job is to reveal defects, find weak points, inconsistent behavior, and circumstances where the software does not work as expected.
- As a tester you need to be comfortable with this role. Given the nature of the tester's tasks, you can see that it is difficult for developers to effectively test their own code Teams of

testers and developers are very common in industry, and projects should have an appropriate developer/tester ratio.

- The ratio will vary depending on available resources, type of project, and TMM level. For example, an embedded realtime system needs to have a lower developer/tester ratio (for example, 2/1) than a simple data base application (4/1 may be suitable).
- At higher TMM levels where there is a well-defined testing group, the developer/ tester ratio would tend to be on the lower end (for example 2/1 versus 4/1) because of the availability of tester resources. In addition to cooperating with code developers, testers also need to work along side with requirements engineers to ensure that requirements are testable, and to plan for system and acceptance test (clients are also involved in the latter).
- Testers also need to work with designers to plan for integration and unit test. In addition, test managers will need to cooperate with project managers in order to develop reasonable test plans, and with upper management to provide input for the development and maintenance of organizational testing standards, polices, and goals.
- Finally, testers also need to cooperate with software quality assurance staff and software engineering process group members. In view of these requirements for multiple working relationships, communication and team working skills are necessary for a successful career as a tester.
- Testers are specialists, their main function is to plan, execute, record, and analyze tests. They do not debug software.
- When defects are detected during testing, software should be returned to the developers who locate the defect and repair the code. The developers have a detailed understanding of the code, and are the best qualified staff to perform debugging.
- Finally, testers need the support of management. Developers, analysts, and marketing staff need to realize that testers add value to a software product in that they detect defects and evaluate quality as early as possible in the software life cycle.
- This ensures that developers release code with few or no defects, and that marketers can deliver software that satisfies the customers' requirements, and is reliable, usable, and correct.
- Test Engineers are usually responsible for:

  Developing test cases and procedures
  - Software testers need to develop test matrices to control the design of test cases.
  - Software Testers need to design test cases based on effective testing techniques.
  - Software testers need to design procedures based on the project needs.

  Test data planning, capture, and conditioning
  - Software testers need to plan test data to be used during test execution.

  Reviewing analysis and design artifacts
  - Software testers need to review and analyze:
  - Requirement documents.
  - Functional Documents.
  - Design Documents.

  Test execution
  - Software testers are responsible for test execution based on testing milestones.

  Utilizing automated test tools for regression testing
  - Software testers are responsible to learn automated testing tools to simplify regression testing.

  Preparing test documentation

  Software testers need to prepare any necessary testware during the project:

- Procedures.
- Guidelines.

Defect tracking and reporting

Software testers are responsible to:

- Find Defects.
- Report defects.
- Verify and validate defect fixes.
- Other testers joining the team will focus on:
- Test execution.
- Defect reporting.
- Regression testing.

The test team should be represented in:

- All key requirements.
- Design meetings, including:
- JAD or requirements definition sessions.
- Risk analysis sessions.
- Prototype review sessions

7. Explain all the defects related to requirement specification, design, coding and testing phases.

Requirement Specification Defect:

The beginning of the software life cycle is critical for ensuring high quality in the software being developed. Defects injected in early phases can persist and be very difficult to remove in later phases. Since many requirements documents are written using a natural language representation, there are very often occurrences of ambiguous, contradictory, unclear, redundant, and imprecise requirements. Specifications in many organizations are also developed using natural language representations, and these too are subject to the same types of problems as mentioned above. However, over the past several years many organizations have introduced the use of formal specification languages that, when accompanied by tools, help to prevent incorrect descriptions of system behavior. Some specific requirements/specification defects are:

1. Functional Description Defects
2. 2. Feature Defects
3. 3. Feature Interaction Defects
4. 4. Interface Description Defects

Design Defects

Design defects occur when system components, interactions between system components, interactions between the components and outside soft ware/hardware, or users are incorrectly designed. This covers defects in the design of algorithms, control, logic, data elements, module interface descriptions, and external software/hardware/user interface descriptions. When describing these defects we assume that the detailed design description for the software modules is at the pseudo code level with processing steps, data structures, input/output parameters, and major control structures defined. If module design is not described in such detail then many of the defects types described here may be moved into the coding defects class.

1. Algorithmic and Processing Defects
2. Control, Logic, and Sequence Defects
3. Data Defects
4. Module Interface Description Defects
5. Functional Description Defects

6. External Interface Description Defects

Coding Defects

Coding defects are derived from errors in implementing the code. Coding defects classes are closely related to design defect classes especially if pseudo code has been used for detailed design. Some coding defects come from a failure to understand programming language constructs, and miscommunication with the designers. Others may have transcription or omission origins. At times it may be difficult to classify a defect as a design or as a coding defect. It is best to make a choice and be consistent when the same defect arises again.

1. Algorithmic and Processing Defects
2. Control, Logic and Sequence Defects
3. Typographical Defects
4. Initialization Defects
5. Data-Flow Defects
6. Data Defects
7. Module Interface Defects
8. Code Documentation Defects
9. External Hardware, Software Interfaces Defects

Testing Defects

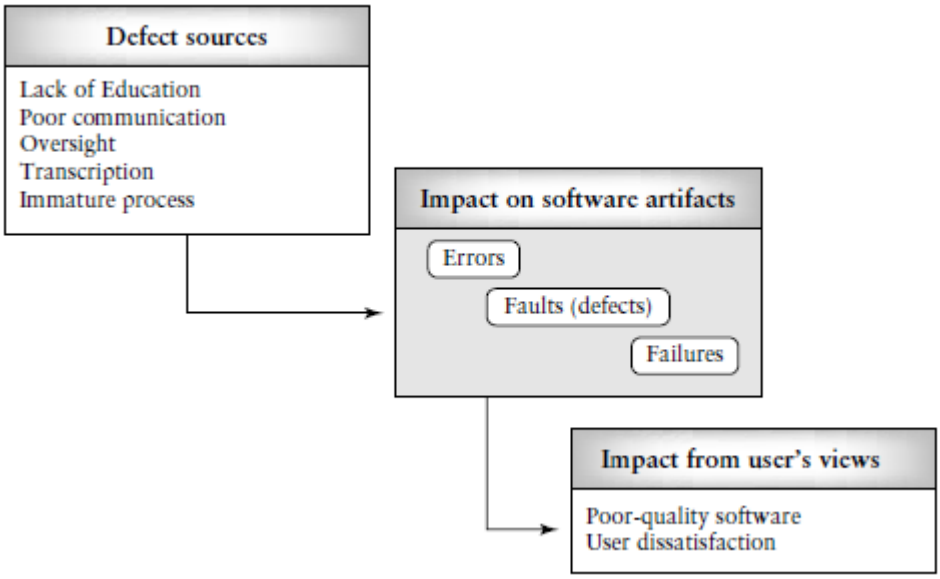
Defects are not confined to code and its related artifacts. Test plans, test cases, test harnesses, and test procedures can also contain defects. Defects in test plans are best detected using review techniques.

1. Test Harness Defects In order to test software, especially at the unit and integration levels, auxiliary code must be developed. This is called the test harness or scaffolding code. Chapter 6 has a more detailed discussion of the need for this code. The test harness code should be carefully designed, implemented, and tested since it a work product and much of this code can be reused when new releases of the software are developed. Test harnesses are subject to the same types of code and design defects that can be found in all other types of software.

2. Test Case Design and Test Procedure Defects These would encompass incorrect, incomplete, missing, inappropriate test cases, and test procedures. These defects are again best detected in test plan reviews as described in Chapter 10. Sometimes the defects are revealed during the testing process itself by means of a careful analysis of test conditions and test results. Repairs will then have to be made.

8. Explain in detail the origin of defects.

The term *defect* and its relationship to the terms *error* and *failure* in the context of the software development domain



1. *Education:* The software engineer did not have the proper educational background to prepare the software artifact. She did not understand how to do something.
For example, a software engineer who did not understand the precedence order of operators in a particular programming language could inject a defect in an equation that uses the operators for a calculation.

2. *Communication:* The software engineer was not informed about something by a colleague.
For example, if engineer 1 and engineer 2 are working on interfacing modules, and engineer 1 does not inform engineer 2 that a no error checking code will appear in the interfacing module he is developing, engineer 2 might make an incorrect assumption relating to the presence/absence of an error check, and a defect will result.

3. *Oversight:* The software engineer omitted to do something. For example, a software engineer might omit an initialization statement.

4. *Transcription:* The software engineer knows what to do, but makes a mistake in doing it. A simple example is a variable name being misspelled when entering the code.

5. *Process:* The process used by the software engineer misdirected her actions. For example, a development process that did not allow sufficient time for a detailed specification to be developed and reviewed could lead to specification defects.

A successful test will reveal the problem and the doctor can begin treatment. Completing the analogy of doctor and ill patient, one could view defective software as the ill patient. Testers as doctors need to have knowledge about possible defects (illnesses) in order to develop defect hypotheses. They use the hypotheses to:

- design test cases;
- design test procedures;
- assemble test sets;
- select the testing levels (unit, integration, etc.) appropriate for the tests;
- evaluate the results of the tests.

Physical defects in the digital world may be due to manufacturing errors, component wear-out, and/or environmental effects.

The fault models are often used to generate a fault list or dictionary. From that dictionary faults can be selected, and test inputs developed for digital components. The effectiveness of a test can be evaluated in the context of the fault model, and is related to the number of faults as expressed in the model, and those actually revealed by the test.

simple example of a fault model a software engineer might have in memory is “an incorrect value for a variable was observed because the precedence order for the arithmetic operators used to calculate its value was incorrect.” This could be called “an incorrect operator precedence order” fault.

An error was made on the part of the programmer who did not understand the order in which the arithmetic operators would execute their operations. Some incorrect assumptions about the order were made.

The defect (fault) surfaced in the incorrect value of the variable. The probable cause is a lack of education on the part of the programmer. Repairs include changing the order of the operators or proper use of parentheses.

The tester with access to this fault model and the frequency of occurrence of this type of fault could use this information as the basis for generating fault hypotheses and test cases.

Unit – II
Part – A

1. What are Black Box Knowledge?

Sources, Requirements, document, Specifications, Domain knowledge, Defect analysis, Data

2. What are Black Box Methods?

Equivalence class, partitioning, Boundary value analysis, State transition testing, Cause and effect graphing, Error guessing

3. What are White box Knowledge sources?

- High-level design
- Detailed design
- Control flow
- graphs
- Cyclomatic
- Complexity

4. What are White box methods?

- Statement testing
- Branch testing
- Path testing
- Data flow testing
- Mutation testing
- Loop testing

5. Define a State.

A state is an internal configuration of a system or component. It is defined in terms of the values assumed at a particular time for the variables that characterize the system or component.

6. Define a finite-state machine.

It is an abstract machine that can be represented by a state graph having a finite number of states and a finite number of transitions between states.

7. Define Random Testing.

Each software module or system has an input domain from which test input data is selected. If a tester randomly selects inputs from the domain, this is called random testing.

8. Define Equivalence Class Partitioning and give its advantages.

If a tester is viewing the software-under-test as a black box with well defined inputs and outputs, a good approach to selecting test inputs is to use a method called equivalence class partitioning. Equivalence class partitioning results in a partitioning of the input domain of the software under test.
Advantages of Equivalence Class Partitioning are,

1. It eliminates the need for exhaustive testing, which is not feasible.

2. It guides a tester in selecting a subset of test inputs with a high probability of detecting a defect.

3. It allows a tester to cover a larger domain of inputs/outputs with a smaller subset selected from an equivalence class.

9. What are the steps involved in Equivalence Class Partitioning?

A good approach includes the following steps.

1. Each equivalence class should be assigned a unique identifier. A simple integer is sufficient.

2. Develop test cases for all valid equivalence classes until all have been covered by (included in) a test case. A given test case may cover more than one equivalence class.

3. Develop test cases for all invalid equivalence classes until all have been covered individually. This is to insure that one invalid case does not mask the effect of another or prevent the execution of another.

10. What are the rules-of-thumb?

The rules-of-thumb described below are useful for getting started with boundary value analysis.

1. If an input condition for the software-under-test is specified as a range of values, develop valid test cases for the ends of the range, and invalid test cases for possibilities just above and below the ends of the range.
2. If an input condition for the software-under-test is specified as a number of values, develop valid test cases for the minimum and maximum numbers as well as invalid test cases that include one lesser and one greater than the maximum and minimum.
3. If the input or output of the software-under-test is an ordered set, such as a table or a linear list, develop tests that focus on the first and last elements of the set.

11. List the steps in developing test cases with a cause-and-effect graph.

1. The tester must decompose the specification of a complex software component into lower-level units.
2. For each specification unit, the tester needs to identify causes and their effects. A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition or a system transformation. Putting together a table of causes and effects helps the tester to record the necessary details. The logical relationships between the causes and effects should be determined. It is useful to express these in the form of a set of rules.
3. From the cause-and-effect information, a Boolean cause-and-effect graph is created. Nodes in the graph are causes and effects. Causes are placed on the left side of the graph and effects on the right. Logical relationships are expressed using standard logical operators such as AND, OR, and NOT, and are associated with arcs.
4. The graph may be annotated with constraints that describe combinations of causes and/or effects that are not possible due to environmental or syntactic constraints.
5. The graph is then converted to a decision table.
6. The columns in the decision table are transformed into test cases.

12. What is State transition testing?

State transition testing is useful for both procedural and object-oriented development. It is based on the concepts of states and finite-state machines, and allows the tester to view the developing software in term of its states, transitions between states, and the inputs and events that trigger state changes.

13. What are the contents of the defect/problem fix report?

The defect/problem fix report should contain the following information:

- project identifier
- the problem/defect identifier
- testing phase that uncovered the defect
- a classification for the defect found
- a description of the repairs that were done
- the identification number(s) of the associated tests
- the date of repair
- the name of the repairer.

14. List the application scope of adequacy criteria.

(i) helping testers to select properties of a program to focus on during test;
(ii) helping testers to select a test data set for a program based on the selected properties;
(iii) supporting testers with the development of quantitative objectives for testing;
(iv) indicating to testers whether or not testing can be stopped for that program.

15. What is test data set?

A test data set is statement, or branch, adequate if a test set T for program P causes all the statements, or branches, to be executed respectively.

16. What is cyclomatic complexity?

A cyclomatic complexity attribute is very useful to a tester The complexity value is usually calculated from the control flow graph (G) by the formula

$V(G) = E - N + 2$

The value E is the number of edges in the control flow graph and N is the number of nodes. This formula can be applied to flow graphs where there are no disconnected components.

17. What are Loop testing strategies?

Loop testing strategies focus on detecting common defects associated with these structures. For example, in a simple loop that can have a range of zero to n iterations, test cases should be developed so that there are:

(i) zero iterations of the loop, i.e., the loop is skipped in its entirely;
(ii) one iteration of the loop;
(iii) two iterations of the loop;
(iv) k iterations of the loop where k _ n;
(v) n _ 1 iterations of the loop;
(vi) n _ 1 iterations of the loop (if possible).

18. What is Mutation testing?

Mutation testing is another approach to test data generation that requires knowledge of code structure, but it is classified as a fault-based testing approach. It considers the possible faults that could occur in a software component as the basis for test data generation and evaluation of testing effectiveness.

19. List the two major assumptions of Mutation testing.

Mutation testing makes two major assumptions:

1. The competent programmer hypothesis. This states that a competent programmer writes programs that are nearly correct. Therefore we can assume that there are no major construction errors in the program; the code is correct except for a simple error(s).

2. The coupling effect. This effect relates to questions a tester might have about how well mutation testing can detect complex errors since the changes made to the code are very simple. DeMillo has commented on that issue as far back as 1978 [10]. He states that test data that can distinguish all programs differing from a correct one only by simple errors are sensitive enough to distinguish it from programs with more complex errors.

20. Define test set.

A test set T is said to be mutation adequate for program P provided that for every inequivalent mutant Pi of P there is an element t in T such that Pi(t) is not equal to P(t).

21. Write down the advantages and disadvantages of random testing.

Advantages of Random Testing :

- Random testing gives us an advantage of easily estimating software reliability from test outcomes.
- They're are done from a user's point of view
- Do not require to know programming languages or how the software has been implemented
- Can be conducted by a body independent from the developers,
- Can be designed as soon as the specifications are complete

Disadvantages of random testing :

- They are not realistic.
- Many of the tests are redundant and unrealistic.
- More time is spent on analysing results.
- One cannot recreate the test if data is not recorded which was used for testing.
- The actual test results are random in the case of randomized software and random testing. Therefore it is not possible to give an exact expected value.

22. What is a control flow graph?

The control flow graph G = (N, E) of a program consists of a set of nodes N and a set of edge E. ν Each node represents a set of program statements. There are five types of nodes. There is a unique entry node and a unique exit node. ν There is an edge from node n1 to node n2 if the control may flow from the last statement in n1 to the first statement in n2 .

23. How to compute cyclomatic complexity?

The cyclomatic complexity of a section of source code is the number of linearly independent paths within it Mathematically, the cyclomatic complexity of a structured program is defined with reference to the control flow graph of the program, a directed graph containing the basic blocks of the program, with an edge between two basic blocks if control may pass from the first to the second. The complexity **M** is then defined as

$M = E - N + 2P$,

where

E = the number of edges of the graph.

N = the number of nodes of the graph.

P = the number of connected components.

24. What is meant by desk checking?

Desk checking is a paper and pencil exercise where the programmer works through the program by hand keeping track of the values of each variable and the statements that are executed.

25. What do you mean by test coverage?

Test coverage measures the amount of testing performed by a set of test. Wherever we can count things and can tell whether or not each of those things has been tested by some test, then we can measure coverage and is known as test coverage.

The basic coverage measure is where the 'coverage item' is whatever we have been able to count and see whether a test has exercised or used this item.

$$\text{Coverage} = \frac{\text{Number of coverage items exercised}}{\text{Total number of coverage items}} \times 100\%$$

Part – B

1. Explain in detail about black box testing strategies and COTS components.

As software development evolves into an engineering discipline, the reuse of software components will play an increasingly important role. Reuse of components means that developers need not reinvent the wheel; instead they can reuse an existing software component with the required functionality. The reusable component may come from a code reuse library within their organization or, as is most likely, from an outside vendor who specializes in the development of specific types of software components. Components produced by vendor organizations are known as commercial off-the-shelf, or COTS, components. The following data illustrate the growing usage of COTS components. In 1997, approximately 25% of the component portfolio of a typical corporation consisted of COTS components. Estimates for 1998 were about 28% and during the next several years the number may rise to 40% .

Using COTS components can save time and money. However, the COTS component must be evaluated before becoming a part of a developing system. This means that the functionality, correctness, and reliability of the component must be established. In addition, its suitability for the application must be determined, and any unwanted functionality must be identified and addressed by the developers. Testing is one process that is not eliminated when COTS components are used for development

When a COTS component is purchased from a vendor it is basically a black box. It can range in size from a few lines of code, for example, a device driver, to thousands of lines of code, as in a telecommunication subsystem. It most cases, no source code is available, and if it is, it is very expensive to purchase. The buyer usually receives an executable version of the component, a description of its functionality, and perhaps a statement of how it was tested. In some cases if the component has been widely adapted, a statement of reliability will also be included. With this limited information, the developers and testers must make a decision on whether or not to use the component. Since the view is mainly as a black box, some of the techniques discussed in this chapter are applicable for testing the COTS components.

If the COTS component is small in size, and a specification of its inputs/outputs and functionality is available, then equivalence class partitioning and boundary value analysis may be useful for detecting defects and establishing component behavior. The tester should also use this approach for identifying any unwanted or unexpected functionality or side effects that could have a detrimental effect on the application. Assertions, which are logic statements that describe correct program behavior, are also useful for assessing COTS behavior

Usage profiles are characterizations of the population of intended uses of the software in its intended environment As in the testing of newly developing software, the testing of COTS components requires the development of test cases, test oracles, and auxiliary code called a test harness. In the case of COTS components, additional code, called glue software, must be developed to bind the COTS component to other modules for smooth system functioning. This glue software must also be tested. All of these activities add to the costs of reuse and must be considered when project plans are developed. Researchers are continually working on issues related to testing and certification of COTS components.

2. Explain cause-and–effect graphing in detail.

Cause-and-effect graphing is a technique that can be used to combine conditions and derive an effective set of test cases that may disclose inconsistencies in a specification. However, the specification must be transformed into a graph that resembles a digital logic circuit. The tester is not required to have a background in electronics, but he should have knowledge of Boolean logic. The graph itself must be expressed in a graphical language.

Developing the graph, especially for a complex module with many combinations of inputs, is difficult and time consuming. The graph must be converted to a decision table that the tester uses to develop test cases. Tools are available for the latter process and allow the derivation of test cases to be more practical using this approach. The steps in developing test cases with a cause-and-effect graph are as follows

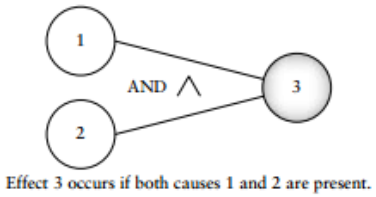
1. The tester must decompose the specification of a complex software component into lower-level units.
2. For each specification unit, the tester needs to identify causes and their effects. A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition or a system transformation. Putting together a table of causes and effects helps the tester to record the necessary details. The logical relationships between the causes and effects should be determined. It is useful to express these in the form of a set of rules.
3. From the cause-and-effect information, a Boolean cause-and-effect graph is created. Nodes in the graph are causes and effects. Causes are placed on the left side of the graph and effects on the right. Logical relationships are expressed using standard logical operators such as AND, OR, and NOT, and are associated with arcs. An example of the notation is shown in Figure 4.4. Myers shows additional examples of graph notations.
4. The graph may be annotated with constraints that describe combinations of causes and/or effects that are not possible due to environmental or syntactic constraints.
5. The graph is then converted to a decision table.
6. The columns in the decision table are transformed into test cases.

The following example illustrates the application of this technique. Suppose we have a specification for a module that allows a user to perform a search for a character in an existing string. The specification states that the user must input the length of the string and the character to search for. If the string length is out-of-range an error message will appear. If the character appears in the string, its position will be reported. If the character is not in the string the message “not found” will be output.

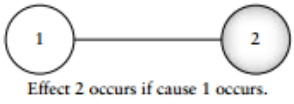
The input conditions, or causes are as follows:

C1: Positive integer from 1 to 80

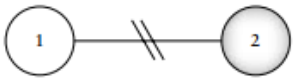
C2: Character to search for is in string



Effect 3 occurs if both causes 1 and 2 are present.



Effect 2 occurs if cause 1 occurs.



Effect 2 occurs if cause 1 does not occur.

The output conditions, or effects are:
E1: Integer out of range
E2: Position of character in string
E3: Character not found

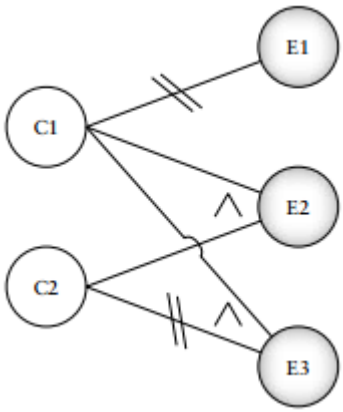
The rules or relationships can be described as follows:
If C1 and C2, then E2.
If C1 and not C2, then E3.
If not C1, then E1.

Based on the causes, effects, and their relationships, a cause-and-effect graph to represent this information is shown in Figure

The next step is to develop a decision table. The decision table reflects the rules and the graph and shows the effects for all possible combinations of causes. Columns list each combination of causes, and each column represents a test case. Given n causes this could lead to a decision table with 2n entries, thus indicating a possible need for many test cases. In this example, since we have only two causes, the size and complexity of the decision table is not a big problem. However, with specifications having large numbers of causes and effects the size of the decision table can be large. Environmental constraints and unlikely combinations may reduce the number of entries and subsequent test cases.

A decision table will have a row for each cause and each effect. The entries are a reflection of the rules and the entities in the cause and effect graph. Entries in the table can be represented by a "1" for a cause or effect that is present, a "0" represents the absence of a cause or effect, and a "—" indicates a "don't care" value. A decision table for our simple example is shown in Table where C1, C2, C3 represent the causes, E1, E2, E3 the effects, and columns T1, T2, T3 the test cases.

The tester can use the decision table to consider combinations of inputs to generate the actual tests. In this example, three test cases are called for. If the existing string is "abcde," then possible tests are the following:



Cause-and-effect graph for the character search example.

Inputs	Length	Character to search for	Outputs
T1	5	c	3
T2	5	w	Not found
T3	90		Integer out of range

One advantage of this method is that development of the rules and the graph from the specification allows a thorough inspection of the specification. Any omissions, inaccuracies, or inconsistencies are likely to be detected. Other advantages come from exercising combinations of test data that may not be considered using other black box testing techniques. The major problem is developing a graph and decision table when there are many causes and effects to consider. A possible solution to this is to decompose a complex specification into lower-level, simpler components and develop cause-and-effect graphs and decision tables for these.

	T1	T2	T3
C1	1	1	0
C2	1	0	—
E1	0	0	1
E2	1	0	0
E3	0	1	0

TABLE 4.3

Decision table for character search example.

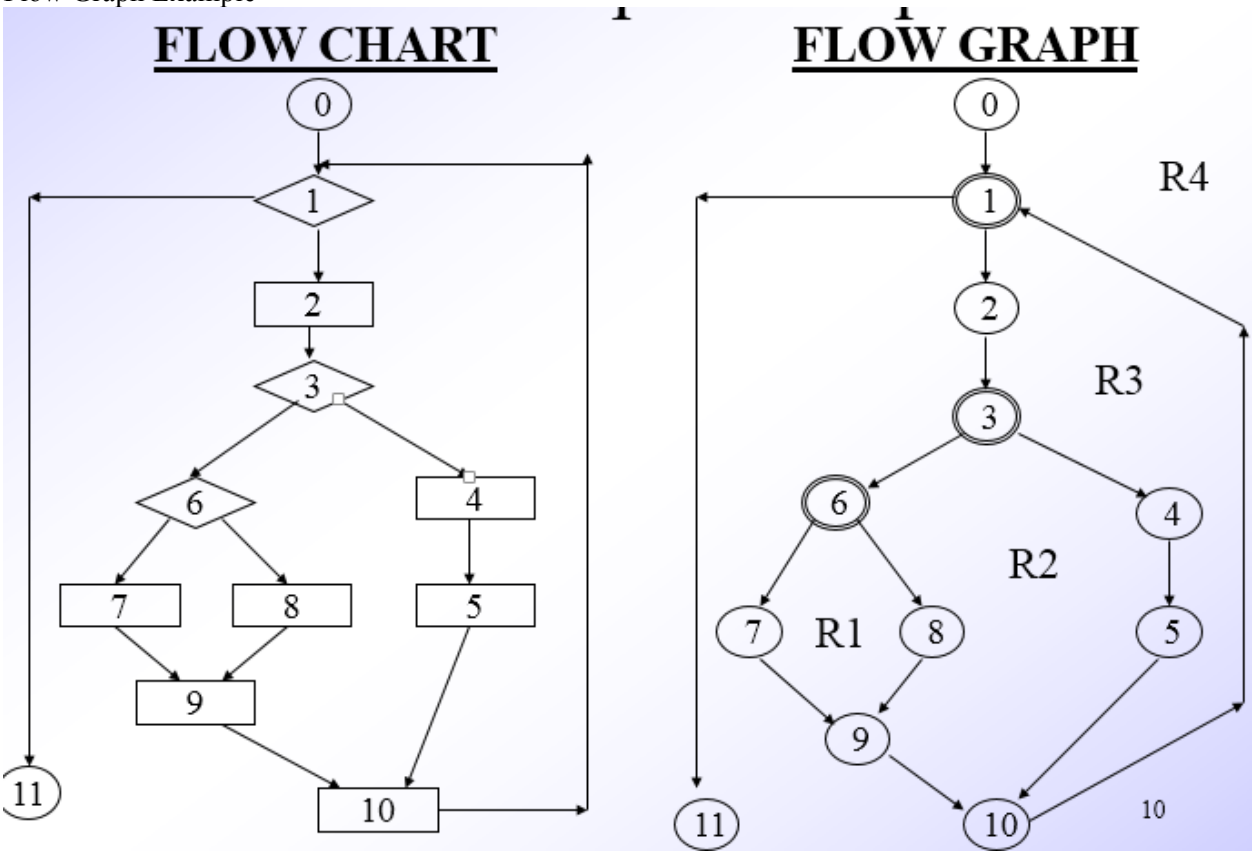
3. What is control flow graph? How is it used in white box test design?

Control flow testing uses the control structure of a program to develop the test cases for the program. ν The test cases are developed to sufficiently cover the whole control structure of the program. The control structure of a program can be represented by the control flow graph of the program.

Flow Graph Notation

- A circle in a graph represents a node, which stands for a sequence of one or more procedural statements
- A node containing a simple conditional expression is referred to as a predicate node
  - Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented by a separate predicate node
  - A predicate node has two edges leading out from it (True and False)
- An edge, or a link, is a an arrow representing flow of control in a specific direction
  - An edge must start and terminate at a node
  - An edge does not intersect or cross over another edge
- Areas bounded by a set of edges and nodes are called regions
- When counting regions, include the area outside the graph as a region, too

Flow Graph Example



Independent Program Paths

- Defined as a path through the program from the start node until the end node that introduces at least one new set of processing statements or a new condition (i.e., new nodes)
- Must move along at least one edge that has not been traversed before by a previous path
- Basis set for flow graph on previous slide
  - Path 1: 0-1-11
  - Path 2: 0-1-2-3-4-5-10-1-11
  - Path 3: 0-1-2-3-6-8-9-10-1-11
  - Path 4: 0-1-2-3-6-7-9-10-1-11
- The number of paths in the basis set is determined by the cyclomatic complexity

Cyclomatic Complexity

- Provides a quantitative measure of the logical complexity of a program
- Defines the number of independent paths in the basis set

- Provides an upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once
- Can be computed three ways
  - The number of regions
  - V(G) = E – N + 2, where E is the number of edges and N is the number of nodes in graph G
  - V(G) = P + 1, where P is the number of predicate nodes in the flow graph G
- Results in the following equations for the example flow graph
  - Number of regions = 4
  - V(G) = 14 edges – 12 nodes + 2 = 4
  - V(G) = 3 predicate nodes + 1 = 4

Deriving the Basis Set and Test Cases

1) Using the design or code as a foundation, draw a corresponding flow graph
2) Determine the cyclomatic complexity of the resultant flow graph
3) Determine a basis set of linearly independent paths
4) Prepare test cases that will force execution of each path in the basis set

4. Explain all additional white box test design approaches?

- Data Flow and White Box Test Design
- Loop Testing
- Mutation Testing

Data Flow and White Box Test Design

In order to discuss test data generation based on data flow information, some basic concepts that define the role of variables in a software component need to be introduced.

We say a variable is defined in a statement when its value is assigned or changed.

For example in the statements

Y = 26 * X

Read (Y)

the variable Y is defined, that is, it is assigned a new value. In data flow notation this is indicated as a def for the variable Y.

We say a variable is used in a statement when its value is utilized in a statement. The value of the variable is not changed.

A more detailed description of variable usage is given by Rapps and Weyuker . They describe a predicate use (p-use) for a variable that indicates its role in a predicate. A computational use (c-use) indicates the variable's role as a part of a computation. In both cases the variable value is unchanged. For example, in the statement Y = 26 * X the variable X is used. Specifically it has a c-use. In the statement if (X >98) Y = max, X has a predicate or p-use. There are other data flow roles for variables such as undefined or dead, . An analysis of data flow patterns for specific variables is often very useful for defect detection. For example, use of a variable without a definition occurring first indicates a defect in the code. The variable has not been initialized. Smart compilers will identify these types of defects. Testers and developers can utilize data flow tools that will identify and display variable role information. These should also be used prior to code reviews to facilitate the work of the reviewers.

Using their data flow descriptions, Rapps and Weyuker identified several data-flow based test adequacy criteria that map to corresponding coverage goals.

- All def
- All p-uses
- All c-uses/some p-uses
- All p-uses/some c-uses

- All uses
- All def-use paths

The strongest of these criteria is all def-use paths. This includes all p- and c-use

```
1       sum = 0                          sum, def
2       read (n),                        n, def
3       i = 1                            i, def
4       while (i <= n)                   i, n p-sue
5             read (number)              number, def
6.            sum = sum + number         sum, def, sum, number, c-use
7             i = i + 1                  i, def, c-use
8       end while
9       print (sum)                      sum, c-use
```

Sample code with data flow information.

To satisfy the all def-use criterion the tester must identify and classify occurrences of all the variables in the software under test. A tabular summary is useful. Then for each variable, test data is generated so that all definitions and all uses for all of the variables are exercised during test.

On the table each def-use pair is assigned an identifier. Line numbers are used to show occurrence of the def or use. Note that in some statements a given variable is both defined and used

Table for n

pair id	def	use
1	2	4

Table for number

pair id	def	use
1	5	6

Table for sum

pair id	def	use
1	1	6
2	1	9
3	6	6
4	6	9

Table for i

pair id	def	use
1	3	4
2	3	7
3	7	7
4	7	4

After completion of the tables, the tester then generates test data to exercise all of these def-use pairs

Test data set 1: $n = 0$

Test data set 2: $n = 5$, number = 1,2,3,4,5

Set 1 covers pair 1 for n, pair 2 for sum, and pair 1 for i. Set 2 covers pair 1 for n, pair 1 for number, pairs 1,3,4 for sum, and pairs 1,2,3,4 for i. Note even for this small piece of code there are four tables and four def-use pairs for two of the variables

Loop Testing

Loops are among the most frequently used control structures. Experienced software engineers realize that many defects are associated with loop constructs. These are often due to poor programming practices and lack of reviews. Therefore, special attention should be paid to loops during testing. Beizer has classified loops into four categories: simple, nested, concatenated, and unstructured

Loop testing strategies focus on detecting common defects associated with these structures. For example, in a simple loop that can have a range of zero to n iterations, test cases should be developed so that there are:

(i) zero iterations of the loop, i.e., the loop is skipped in its entirely;
(ii) one iteration of the loop;
(iii) two iterations of the loop;
(iv) k iterations of the loop where k < n;
(v) n - 1 iterations of the loop;
(vi) n +1 iterations of the loop (if possible).

If the loop has a nonzero minimum number of iterations, try one less than the minimum. Other cases to consider for loops are negative values for the loop control variable, and n 1 iterations of the loop if that is possible. Zhu has described a historical loop count adequacy criterion that states that in the case of a loop having a maximum of n iterations, tests that execute the loop zero times, once, twice, and so on up to n times are required

Mutation Testing

Mutation testing is another approach to test data generation that requires knowledge of code structure, but it is classified as a fault-based testing approach. It considers the possible faults that

could occur in a software component as the basis for test data generation and evaluation of testing effectiveness.

Mutation testing makes two major assumptions:

1. The competent programmer hypothesis. This states that a competent programmer writes programs that are nearly correct. Therefore we can assume that there are no major construction errors in the program; the code is correct except for a simple error(s).
2. The coupling effect. This effect relates to questions a tester might have about how well mutation testing can detect complex errors since the changes made to the code are very simple.

DeMillo states that test data that can distinguish all programs differing from a correct one only by simple errors are sensitive enough to distinguish it from programs with more complex errors. Mutation testing starts with a code component, its associated test cases, and the test results. The original code component is modified in a simple way to provide a set of similar components that are called mutants.

Each mutant contains a fault as a result of the modification. The original test data is then run with the mutants. If the test data reveals the fault in the mutant (the result of the modification) by producing a different output as a result of execution, then the mutant is said to be killed. If the mutants do not produce outputs that differ from the original with the test data, then the test data are not capable of revealing such defects. The tests cannot distinguish the original from the mutant. The tester then must develop additional test data to reveal the fault and kill the mutants.

Mutations are simple changes in the original code component, for example: constant replacement, arithmetic operator replacement, data statement alteration, statement deletion, and logical operator replacement. There are existing tools that will easily generate mutants. Tool users need only to select a change operator

To measure the mutation adequacy of a test set T for a program P we can use what is called a mutation score (MS), which is calculated as follows

$$\text{MS}\ (P,T) = \frac{\#\ \text{of dead mutants}}{\#\ \text{total mutants} - \#\ \text{of equivalent mutants}}$$

Mutation testing is useful in that it can show that certain faults as represented in the mutants are not likely to be present since they would have been revealed by test data. It also helps the tester to generate hypotheses about the different types of possible faults in the code and to develop test cases to reveal them.

5. How white box testing methods related to TMM?

White box methods also provide a systematic way of developing test cases. However, white box methods have a stronger theoretical foundation and are supported by test adequacy criteria that guide the development of test data and allow evaluation of testing goals when tests are subsequently executed. In addition, white box methods have adequate tool support, and they depend on the use of notations that allow a connection to other development activities, for example, design.

Managers should ensure that testers are trained in the use of both for consistent application to all organizational projects as described in the TMM. The Activities/Tasks/Responsibilities (ATR's) associated with adapting and implementing black box methods also apply to white box methods as well.

When making a choice among white box testing methods the tester must consider the nature of the software to be tested, resources available, and testing constraints and conditions. For example, a tester might choose to develop test designs using elements of control flow such as branches. In this same example, to insure coverage of compound conditions the tester may decide that multiple decision coverage is a wise testing goal. However, if the code is not complex, and is not mission, safety, or business critical, then simple branch coverage might be sufficient. The tester must also apply this reasoning to selection of a degree of coverage. For example, for a simple non mission

critical module, 85% branch coverage may prove to be a sufficient goal for the module if testing resources are tight

In all cases the tester should select a combination of strategies to develop test cases that includes both black box and white box approaches. No one test design approach is guaranteed to reveal all defects, no matter what its proponents declare! Use of different testing strategies and methods has the following benefits

1. The tester is encouraged to view the developing software from several different views to generate the test data. The views include control flow, data flow, input/output behavior, loop behavior, and states/state changes. The combination of views, and the test cases developed from their application, is more likely to reveal a wide variety of defects, even those that are difficult to detect. This results in a higher degree of software quality
2. The tester must interact with other development personnel such as requirements analysts and designers to review their representations of the software. Representations include input/output specifications, pseudo code, state diagrams, and control flow graphs which are rich sources for test case development. As a result of the interaction, testers are equipped with a better understanding of the nature of the developing software, can evaluate its testability, give intelligent input during reviews, generate hypotheses about possible defects, and develop an effective set of tests.
3. The tester is better equipped to evaluate the quality of the testing effort (there are more tools and approaches available from the combination of strategies). The testers are also more able to evaluate the quality of the software itself, and establish a high degree of confidence that the software is operating occurring to the specifications. This higher confidence is a result of having examined software behavior and structural integrity in several independent ways.
4. The tester is better able to contribute to organizational test process improvement efforts based on his/her knowledge of a variety of testing strategies. With a solid grasp of both black and white box test design strategies, testers can have a very strong influence on the development and maintenance of test policies, test plans, and test practices. Testers are also better equipped to fulfill the requirements for the Activities, Tasks, and Responsibilities called for at TMM level. With their knowledge they can promote best practices, technology transfer, and ensure organization wide adaptation of a variety of test design strategies and techniques.

6. Explain in detail about Equivalence Class Partitioning.

Equivalence class partitioning results in a partitioning of the input domain of the software under-test. The technique can also be used to partition the output domain, but this is not a common usage. The finite number of partitions or equivalence classes that result allow the tester to select a given member of an equivalence class as a representative of that class. It is assumed that all members of an equivalence class are processed in an equivalent way by the target software.

Using equivalence class partitioning a test value in a particular class is equivalent to a test value of any other member of that class. Therefore, if one test case in a particular equivalence class reveals a defect, all the other test cases based on that class would be expected to reveal the same defect. We can also say that if a test case in a given equivalence class did not detect a particular type of defect, then no other test case based on that class would detect the defect (unless a subset of the equivalence class falls into another equivalence class, since classes may overlap in some cases). A more formal discussion of equivalence class partitioning is given in Beizer

Based on this discussion of equivalence class partitioning we can say that the partitioning of the input domain for the software-under-test using this technique has the following advantages:

1. It eliminates the need for exhaustive testing, which is not feasible.
2. It guides a tester in selecting a subset of test inputs with a high probability of detecting a defect.

3. It allows a tester to cover a larger domain of inputs/outputs with a smaller subset selected from an equivalence class

Important points related to equivalence class partitioning

1. The tester must consider both valid and invalid equivalence classes. Invalid classes represent erroneous or unexpected inputs.
2. Equivalence classes may also be selected for output conditions.
3. The derivation of input or outputs equivalence classes is a heuristic process. The conditions that are described in the following paragraphs only give the tester guidelines for identifying the partitions. There are no hard and fast rules. Given the same set of conditions, individual testers may make different choices of equivalence classes. As a tester gains experience he is more able to select equivalence classes with confidence.
4. In some cases it is difficult for the tester to identify equivalence classes. The conditions/boundaries that help to define classes may be absent, or obscure, or there may seem to be a very large or very small number of equivalence classes for the problem domain. These difficulties may arise from an ambiguous, contradictory, incorrect, or incomplete specification and/or requirements description. It is the duty of the tester to seek out the analysts and meet with them to clarify these documents.

List of Conditions

1. ''If an input condition for the software-under-test is specified as a range of values, select one valid equivalence class that covers the allowed range and two invalid equivalence classes, one outside each end of the range.''
2. ''If an input condition for the software-under-test is specified as a number of values, then select one valid equivalence class that includes the allowed number of values and two invalid equivalence classes that are outside each end of the allowed number.''
3. ''If an input condition for the software-under-test is specified as a set of valid input values, then select one valid equivalence class that contains all the members of the set and one invalid equivalence class for any value outside the set.''
4. ''If an input condition for the software-under-test is specified as a "must be" condition, select one valid equivalence class to represent the "must be" condition and one invalid class that does not include the "must be" condition.''

7. Explain State transition testing and error guessing.

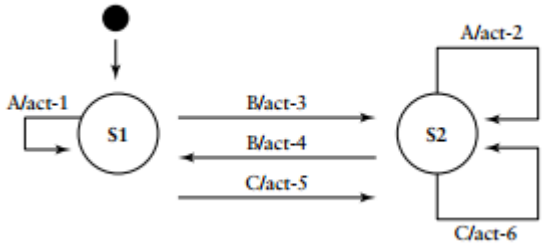
State transition testing is useful for both procedural and object-oriented development. It is based on the concepts of states and finite-state machines, and allows the tester to view the developing software in term of its states, transitions between states, and the inputs and events that trigger state changes. This view gives the tester an additional opportunity to develop test cases to detect defects that may not be revealed using the input/output condition as well as cause-and-effect views presented by equivalence class partitioning and cause-and-effect graphing.

The data flow approach is most effective at the unit level of testing. When code becomes more complex and there are more variables to consider it becomes more time consuming for the tester to analyze data flow roles, identify paths, and design the tests

A state is an internal configuration of a system or component. It is defined in terms of the values assumed at a particular time for the variables that characterize the system or component. A finite-state machine is an abstract machine that can be represented by a state graph having a finite number of states and a finite number of transitions between states. During the specification phase a state transition graph (STG) may be generated for the system as a whole and/or specific modules. In object oriented development the graph may be called a state chart. STG/state charts are useful models of software (object) behavior. STG/state charts are commonly depicted by a set of nodes (circles, ovals, rounded rectangles) which represent states. These usually will have a name or number to

identify the state. A set of arrows between nodes indicate what inputs or events will cause a transition or change between the two linked states.

Outputs/actions occurring with a state transition are also depicted on a link or arrow. A simple state transition diagram is shown in Figure . S1 and S2 are the two states of interest. The black dot represents a pointer to the initial state from outside the machine. Many STGs also have "error" states and "done" states, the latter to indicate a final state for the system. The arrows display inputs/actions that cause the state transformations in the arrow directions. For example, the transition from S1 to S2 occurs with input, or event B. Action 3 occurs as part of this state transition. This is represented by the symbol "B/act3."



It is often useful to attach to the STG the system or component variables that are affected by state transitions. This is valuable information for the tester as we will see in subsequent paragraphs. For large systems and system components, state transition graphs can become very complex. Developers can nest them to represent different levels of abstraction. This approach allows the STG developer to group a set of related states together to form an encapsulated state that can be represented as a single entity on the original STG. The STG developer must ensure that this new state has the proper connections to the unchanged states from the original STG. Another way to simplify the STG is to use a state table representation which may be more concise. A state table for the STG in Figure is shown in Table

	S1	S2
Inputs		
Input A	S1 (act-1)	S2 (act-2)
Input B	S2 (act-3)	S1 (act-4)
Input C	S2 (act-5)	S2 (act-6)

The state table lists the inputs or events that cause state transitions. For each state and each input the next state and action taken are listed. Therefore, the tester can consider each entity as a representation of a state transition.

The STGs should be subject to a formal inspection when the requirement/specification is reviewed. This step is required for organization assessed at TMM level 3 and higher. It is essential that

testers be present at the reviews. From the tester's view point the review should ensure that (i) the proper number of states are represented, (ii) each state transition (input/output/action) is correct, (iii) equivalent states are identified, and (iv) unreachable and dead states are identified. Unreachable states are those that no input sequence will reach, and may indicate missing transitions. Dead states are those that once entered cannot be exited. In rare cases a dead state is legitimate, for example, in software that controls a destructible device.

After the STG has been reviewed formally the tester should plan appropriate test cases. An STG has similarities to a control flow graph in that it has paths, or successions of transitions, caused by a sequence of inputs. Coverage of all paths does not guarantee complete testing and may not be

practical. A simple approach might be to develop tests that insure that all states are entered. A more practical and systematic approach suggested by Marik consists of testing every possible state transition

Error Guessing

Designing test cases using the error guessing approach is based on the tester's/developer's past experience with code similar to the code-under test, and their intuition as to where defects may lurk in the code. Code similarities may extend to the structure of the code, its domain, the design approach used, its complexity, and other factors. The tester/developer is sometimes able to make an educated "guess" as to which types of defects may be present and design test cases to reveal them. Some examples of obvious types of defects to test for are cases where there is a possible division by zero, where there are a number of pointers that are manipulated, or conditions around array boundaries. Error guessing is an ad hoc approach to test design in most cases. However, if defect data for similar code or past releases of the code has been carefully recorded, the defect types classified, and failure symptoms due to the defects carefully noted, this approach can have some structure and value. Such data would be available to testers in a TMM level 4 organization.

8. How will you use white box approach to test case design?

White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing). In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. This is analogous to testing nodes in a circuit, e.g. in-circuit testing (ICT). White-box testing can be applied at the unit, integration and system levels of the software testing process. Although traditional testers tended to think of white-box testing as being done at the unit level, it is used for integration and system testing more frequently today. It can test paths within a unit, paths between units during integration, and between subsystems during a system–level test.

White-box testing's basic procedures involves the tester having a deep level of understanding of the source code being tested. The programmer must have a deep understanding of the application to know what kinds of test cases to create so that every visible path is exercised for testing. Once the source code is understood then the source code can be analyzed for test cases to be created. These are the three basic steps that white-box testing takes in order to create test cases:

1. Input involves different types of requirements, functional specifications, detailed designing of documents, proper source code, security specifications. This is the preparation stage of white-box testing to layout all of the basic information.
2. Processing involves performing risk analysis to guide whole testing process, proper test plan, execute test cases and communicate results. This is the phase of building test cases to make sure they thoroughly test the application the given results are recorded accordingly.
3. Output involves preparing final report that encompasses all of the above preparations and results

Unit – III
Part – A

1. What are the levels or major phases of testing?

Unit test, integration test, system test, and acceptance test

2. Define Unit.

A unit is the smallest possible testable software component. It can be characterized in several ways. For example, a unit in a typical procedure-oriented software system:

- performs a single cohesive function;
- can be compiled separately;
- is a task in a work breakdown structure (from the manager's point of view);
- contains code that can fit on a single page or screen.

3. What are the tasks to be performed in unit test?

To prepare for unit test the developer/tester must perform several tasks. These are:

(i) plan the general approach to unit testing;

(ii) design the test cases, and test procedures (these will be attached to the test plan);

(iii) define relationships between the tests;

(iv) prepare the auxiliary code necessary for unit test.

4. Define test harness.

The auxiliary code developed to support testing of units and components is called a test harness. The harness consists of drivers that call the target code and stubs that represent modules it calls.

5. What are the goals of Integration test?

Integration test for procedural code has two major goals:

(i) to detect defects that occur on the interfaces of units;

(ii) to assemble the individual units into working subsystems and finally a complete system that is ready for system test.

6. Define Cluster.

A cluster consists of classes that are related, for example, they may work together (cooperate) to support a required functionality for the complete system.

7. What are the items included in a Cluster Test Plan?

Cluster Test Plan includes the following items:

(i) clusters this cluster is dependent on;

(ii) a natural language description of the functionality of the cluster to be tested;

(iii) list of classes in the cluster;

(iv) a set of cluster test cases.

8. List the types of system test.

There are several types of system tests as follows:

- Functional testing
- Performance testing
- Stress testing
- Configuration testing
- Security testing
- Recovery testing

9. Define Functional Testing.

Functional tests are black box in nature. The focus is on the inputs and proper outputs for each function. Improper and illegal inputs must also be handled by the system. System behavior under the latter circumstances tests must be observed. All functions must be tested.

10. What are the goals of the Functional Testing?

- All types or classes of legal inputs must be accepted by the software.
- All classes of illegal inputs must be rejected.
- All possible classes of system output must exercise and examined.
- All effective system states and state transitions must be exercised and examined.
- All functions must be exercised.

11. What are two major types of requirements?

1. Functional requirements. Users describe what functions the software should perform. We test for compliance of these requirements at the system level with the functional-based system tests.

2. Quality requirements. There are nonfunctional in nature but describe quality levels expected for the software. One example of a quality requirement is performance level. The users may have objectives for the software system in terms of memory use, response time, throughput, and delays.

12. What are the goals of system performance test?

The goal of system performance tests is to see if the software meets the performance requirements. Testers also learn from performance test whether there are any hardware or software factors that impact on the system's performance. Performance testing allows testers to tune the system; that is, to optimize the allocation of system resources.

13. Define Stress Testing.

When a system is tested with a load that causes it to allocate its resources in maximum amounts, this is called stress testing. For example, if an operating system is required to handle 10 interrupts/second and the load causes 20 interrupts/second, the system is being stressed.

14. Define Configuration testing.

Configuration testing allows developers/testers to evaluate system performance and availability when hardware exchanges and reconfigurations occur. Configuration testing also requires many resources including the multiple hardware devices used for the tests. If a system does not have specific requirements for device configuration changes then large-scale configuration testing is not essential.

15. What are the objectives of configuration testing according to Beizer?

• Show that all the configuration changing commands and menus work properly.

• Show that all interchangeable devices are really interchangeable, and that they each enter the proper states for the specified conditions.

• Show that the systems' performance level is maintained when devices are interchanged, or when they fail.

16. What is security testing?

Security testing evaluates system characteristics that relate to the availability, integrity, and confidentially of system data and services. Users/clients should be encouraged to make sure their security needs are clearly known at requirements time, so that security issues can be addressed by designers and testers.

17. What are the areas to be focused on during security testing?

- Password Checking
- Password Expiration
- Encryption.
- Legal and Illegal Entry with Passwords
- Browsing
- Trap Doors
- Viruses

18. Define Recovery testing.

Recovery testing subjects a system to losses of resources in order to determine if it can recover properly from these losses. This type of testing is especially important for transaction systems, for example, on-line banking software.

19. Define Regression testing.

Regression testing is not a level of testing, but it is the retesting of software that occurs when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new defects have been introduced due to the changes. Regression testing can occur at any level of test.

20. Define a use case.

A use case is a pattern, scenario, or exemplar of usage. It describes a typical interaction between the software system under development and a user.

21. Define internationalization testing.

Internationalization testing is the process of verifying the application under test to work uniformly across multiple regions and cultures.

The main purpose of internationalization is to check if the code can handle all international support without breaking functionality that might cause data loss or data integrity issues. Globalization testing verifies if there is proper functionality of the product with any of the locale settings.

22. Define scenario testing.

Scenario testing is a software testing activity that uses scenarios: hypothetical stories to help the tester work through a complex problem or test system. The ideal scenario test is a credible, complex, compelling or motivating story the outcome of which is easy to evaluate.

23. What is localization testing? List its characteristics.

Localization testing is performed to verify the quality of a product's localization for a particular target culture/locale and is executed only on the localized version of the product.

Localization Testing - Characteristics:

- Modules affected by localization, such as UI and content
- Modules specific to Culture/locale-specific, language-specific, and region-specific
- Critical Business Scenarios Testing
- Installation and upgrading tests run in the localized environment
- Plan application and hardware compatibility tests according to the product's target region.

24. Define acceptance testing.

Acceptance testing, a testing technique performed to determine whether or not the software system has met the requirement specifications. The main purpose of this test is to evaluate the system's compliance with the business requirements and verify if it is has met the required criteria for delivery to end users.

There are various forms of acceptance testing:

- User acceptance Testing
- Business acceptance Testing
- Alpha Testing
- Beta Testing

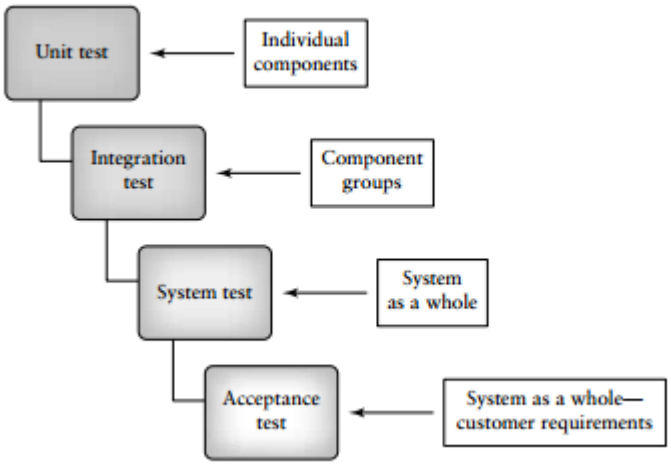
25. Define alpha and beta testing.

Alpha testing takes place at the developer's site by the internal teams, before release to external customers. This testing is performed without the involvement of the development teams.

Beta testing also known as user testing takes place at the end users site by the end users to validate the usability, functionality, compatibility, and reliability testing.

Part – B

1. Explain the levels of testing in detail?



The approach used to design and develop a software system has an impact on how testers plan and design suitable tests. There are two major approaches to system development—bottom-up, and top-down. These approaches are supported by two major types of programming languages—procedure-oriented and object-oriented. Testing at different levels for systems are developed with both approaches using either traditional procedural programming languages or object-oriented programming languages. The different nature of the code produced requires testers to use different strategies to identify and test components and component groups.

Systems developed with procedural languages are generally viewed as being composed of passive data and active procedures. When test cases are developed the focus is on generating input data to pass to the procedures (or functions) in order to reveal defects. Object oriented systems are viewed as being composed of active data along with allowed operations on that data, all encapsulated within a unit similar to an abstract data type. The operations on the data may not be called upon in any specific order. Testing this type of software means designing an order of calls to the operations using various parameter values in order to reveal defects. Issues related to inheritance of operations also impact on testing.

Levels of abstraction for the two types of systems are also somewhat different. In traditional procedural systems, the lowest level of abstraction is described as a function or a procedure that performs some simple task. The next higher level of abstraction is a group of procedures (or functions) that call one another and implement a major system requirement. These are called subsystems. Combining subsystems finally produces the system as a whole, which is the highest level of abstraction. In object-oriented systems the lowest level is viewed by some researchers as the method or member function .The next highest level is viewed as the class that encapsulates data and methods that operate on the data. To move up one more level in an object-oriented system some researchers use the concept of the cluster, which is a group of cooperating or related classes. Finally, there is the system level, which is a combination of all the clusters and any auxiliary code needed to run the system.

Not all researchers in object-oriented development have the same view of the abstraction levels, for example, Jorgensen describes the thread as a highest level of abstraction. Differences of opinion will be described in other sections of this chapter. While approaches for testing and assembling traditional procedural type systems are well established, those for object-oriented systems are still the subject of ongoing research efforts. There are different views on how unit, integration, and system

tests are best accomplished in object oriented systems. When object-oriented development was introduced key beneficial features were encapsulation, inheritance, and polymorphism. It was said that these features would simplify design and development and encourage reuse.

However, testing of object-oriented systems is not straightforward due to these same features. For example, encapsulation can hide details from testers, and that can lead to uncovered code. Inheritance also presents many testing challenges, among those the retesting of inherited methods when they are used by a subclass in a different context. It is also difficult to define a unit for object-oriented code. Some researchers argue for the method (member function) as the unit since it is procedure like. However, some methods are very small in size, and developing test harnesses to test each individually requires a large overhead. Should a single class be a unit? If so, then the tester need to consider the complexity of the test harness needed to test the unit since in many cases, a particular class depends on other classes for its operation. Also, object-oriented code is characterized by use of messages, dynamic binding, state changes, and nonhierarchical calling relationships. This also makes testing more complex.

2. Explain Unit Testing.

A workable definition for a software unit is as follows: A unit is the smallest possible testable software component. It can be characterized in several ways. For example, a unit in a typical procedure-oriented software system: • performs a single cohesive function; • can be compiled separately; • is a task in a work breakdown structure (from the manager's point of view); • contains code that can fit on a single page or screen. A unit is traditionally viewed as a function or procedure implemented in a procedural (imperative) programming language. In object-oriented systems both the method and the class/object have been suggested by researchers as the choice for a unit. The relative merits of each of these as the selected component for unit test are described in sections that follow. A unit may also be a small-sized COTS component purchased from an outside vendor that is undergoing evaluation by the purchaser

Components suitable for unit testing or a simple module retrieved from an in-house reuse library. These unit types are shown in Figure

Unit Test: The Need for Preparation:

The principal goal for unit testing is insure that each individual software unit is functioning according to its specification. Good testing practice calls for unit tests that are planned and public. Planning includes designing tests to reveal defects such as functional description defects, algorithmic defects, data defects, and control logic and sequence defects. Resources should be allocated, and test cases should be developed, using both white and black box test design strategies. The unit should be tested by an independent tester (someone other than the developer) and the test results and defects found should be recorded as a part of the unit history (made public). Each unit should also be reviewed by a team of reviewers, preferably before the unit test. Unfortunately, unit test in many cases is performed informally by the unit developer soon after the module is completed, and it compiles cleanly.

Some developers also perform an informal review of the unit. Under these circumstances the review and testing approach may be ad hoc. Defects found are often not recorded by the

developer; they are private (not public), and do not become a part of the history of the unit. This is poor practice, especially if the unit performs mission or safely critical tasks, or is intended for reuse.

To implement best practices it is important to plan for, and allocate resources to test each unit. If defects escape detection in unit test because of poor unit testing practices, they are likely to show up during integration, system, or acceptance test where they are much more costly to locate and repair. In the worst-case scenario they will cause failures during operation requiring the development organization to repair the software at the clients' site. This can be very costly.

To prepare for unit test the developer/tester must perform several tasks. These are:

(iii) plan the general approach to unit testing;
(ii) design the test cases, and test procedures (these will be attached to the test plan);
(iii) define relationships between the tests;
(iv) prepare the auxiliary code necessary for unit test

Unit Test Planning

A general unit test plan should be prepared. It may be prepared as a component of the master test plan or as a stand-alone plan. It should be developed in conjunction with the master test plan and the project plan for each project. Documents that provide inputs for the unit test plan are the project plan, as well the requirements, specification, and design documents that describe the target units. Components of a unit test plan are described in detail the IEEE Standard for Software Unit Testing

A brief description of a set of development phases for unit test planning is found below

Phase 1: Describe Unit Test Approach and Risks

In this phase of unit testing planning the general approach to unit testing is outlined.

The test planner: (i) identifies test risks;

(iv) describes techniques to be used for designing the test cases for the units;

(v) describes techniques to be used for data validation and recording of test results; (iv) describes the requirements for test harnesses and other software that interfaces with the units to be tested, for example, any special objects needed for testing object-oriented units.

Phase 2: Identify Unit Features to be Tested

This phase requires information from the unit specification and detailed design description. The planner determines which features of each unit will be tested, for example: functions, performance requirements, states, and state transitions, control structures, messages, and data flow patterns. If some features will not be covered by the tests, they should be mentioned and the risks of not testing them be assessed. Input/output characteristics associated with each unit should also be identified, such as variables with an allowed ranges of values and performance at a certain level.

Phase 3: Add Levels of Detail to the Plan

In this phase the planner refines the plan as produced in the previous two phases. The planner adds new details to the approach, resource, and scheduling portions of the unit test plan. As an example, existing test cases that can be reused for this project can be identified in this phase. Unit availability and integration scheduling information should be included in the revised version of the test plan. The planner must be sure to include a description of how test results will be recorded. Test-related documents that will be required for this task, for example, test logs, and test incident reports, should be described, and references to standards for these documents provided. Any special tools required for the tests are also described. The next steps in unit testing consist of designing the set of test cases, developing the auxiliary code needed for testing, executing the tests, and recording and analyzing the results

Designing the Unit Tests

Part of the preparation work for unit test involves unit test design. It is important to specify (i) the test cases (including input data, and expected outputs for each test case), and, (ii) the test procedures (steps required run the tests).

As part of the unit test design process, developers/testers should also describe the relationships between the tests. Test suites can be defined that bind related tests together as a group. All of this test design information is attached to the unit test plan. Test cases, test procedures, and test suites may be reused from past projects if the organization has been careful to store them so that they are easily retrievable and reusable. Test case design at the unit level can be based on use of the black and white box test design strategies. Both of these approaches are useful for designing test cases for functions and procedures. They are also useful for designing tests for the individual methods (member functions) contained in a class.

3. **Summarize the issues that arise in class testing**.

Class Testing:

The complete test coverage of a class involves

- Testing all operations associated with an object
- Setting and interrogating all object attributes
- Exercising the object in all possible states

Preparatory steps for class testing

(i) List specified states for the object that is to be tested
(ii) List the messages and operations that will be exercised as a consequence of the test
(iii) List the exceptions that may occur as the object is tested
(iv) List external conditions
(v) Document all supplementary information that will aid in understanding or implementing the test

A class cannot be tested directly (i.e.) a class can be tested only indirectly by testing several of its instances.

Issue: How can an abstract class be tested which has no instance.

Control flow is characterized by message passing amongst the objects

Issue: There is no sequential control flow within a class

Inheritance introduces new issues in object oriented approach

Issue: Structure of Inheritance e.g. possibility of repeated or multiple inheritance makes an object more error prone

Type of inheritance , e.g. if derived class is redefined

To test an object, interaction between the methods provided by the object need to be tested.

Abstract classes may be tested by using the approach of using different patterns foe invoking the methods.

4. Explain in detail about test harness.

In addition to developing the test cases, supporting code must be developed to exercise each unit and to connect it to the outside world. Since the tester is considering a stand-alone function/procedure/class, rather than a complete system, code will be needed to call the target unit, and also to represent modules that are called by the target unit. This code called the test harness, is

developed especially for test and is in addition to the code that composes the system under development

The auxiliary code developed to support testing of units and components is called a test harness. The harness consists of drivers that call the target code and stubs that represent modules it calls

The development of drivers and stubs requires testing resources. The drivers and stubs must be tested themselves to insure they are working properly and that they are reusable for subsequent releases of the software. Drivers and stubs can be developed at several levels of functionality. For example, a driver could have the following options and combinations of options:

5. call the target unit;

(ii) do 1, and pass inputs parameters from a table;

(iii) do 1, 2, and display parameters;

(iv) do 1, 2, 3 and display results (output parameters).

The stubs could also exhibit different levels of functionality. For example a stub could:

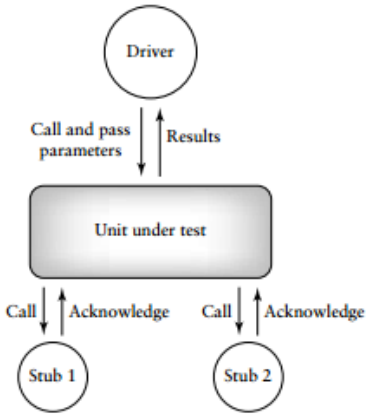
(i) display a message that it has been called by the target unit;

(ii) do 1, and display any input parameters passed from the target unit;

(iii) do 1, 2, and pass back a result from a table;

(iv) do 1, 2, 3, and display result from table

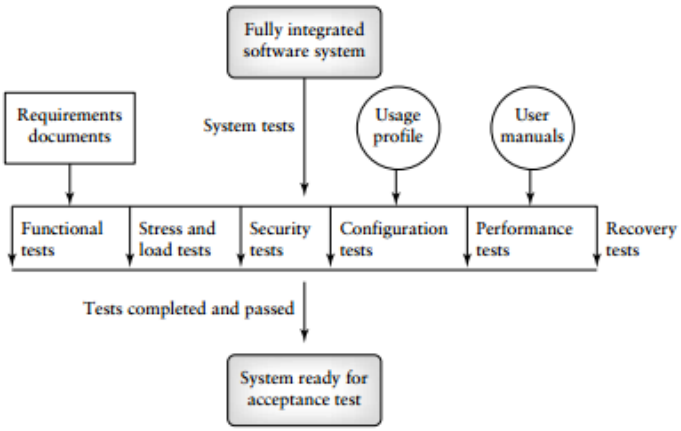
Test harness



Drivers and stubs as shown in Figure are developed as procedures and functions for traditional imperative-language based systems. For object-oriented systems, developing drivers and stubs often means the design and implementation of special classes to perform the required testing tasks. The test harness itself may be a hierarchy of classes. For example, in Figure the driver for a procedural system may be designed as a single procedure or main module to call the unit under test; however, in an object-oriented system it may consist of several test classes to emulate all the classes that call for services in the class under test. Researchers such as Rangaraajan and Chen have developed tools that generate test cases using several different approaches, and classes of test harness objects to test object-oriented code.

The test planner must realize that, the higher the degree of functionally for the harness, the more resources it will require to design, implement, and test. Developers/testers will have to decide depending on the nature of the code under test, just how complex the test harness needs to be. Test harnesses for individual classes tend to be more complex than those needed for individual procedures and functions since the items being tested are more complex and there are more interactions to consider.

5. Explain different types of system testing.



The types are as follows:

Functional testing

Functional tests at the system level are used to ensure that the behavior of the system adheres to the requirements specification. All functional requirements for the system must be achievable by the system.

Functional tests are black box in nature. The focus is on the inputs and proper outputs for each function. Improper and illegal inputs must also be handled by the system. System behavior under the latter circumstances tests must be observed. All functions must be tested that can be used to generate test cases. State-based tests are also valuable. In fact, the tests should focus on the following goals.

- All types or classes of legal inputs must be accepted by the software.
- All classes of illegal inputs must be rejected (however, the system should remain available).
- All possible classes of system output must exercised and examined.
- All effective system states and state transitions must be exercised and examined.
- All functions must be exercised

Performance testing

The goal of system performance tests is to see if the software meets the performance requirements. Testers also learn from performance test whether there are any hardware or software factors that impact on the system's performance. Performance testing allows testers to tune the system; that is, to optimize the allocation of system resources. For example, testers may find that they need to reallocate memory pools, or to modify the priority level of certain system operations. Testers may also be able to project the system's future performance levels. This is useful for planning subsequent releases.

Stress testing

When a system is tested with a load that causes it to allocate its resources in maximum amounts, this is called stress testing

Stress testing is important because it can reveal defects in real-time and other types of systems, as well as weak areas where poor design could cause unavailability of service. For example, system prioritization orders may not be correct, transaction processing may be poorly designed and waste memory space, and timing sequences may not be appropriate for the required tasks. This is particularly important for real-time systems where unpredictable events may occur resulting in input loads that exceed those described in the requirements documents. Stress testing often uncovers race conditions, deadlocks, depletion of resources in unusual or unplanned patterns, and upsets in normal operation of the software system.

Configuration testing

According to Beizer configuration testing has the following objectives

- Show that all the configuration changing commands and menus work properly.

• Show that all interchangeable devices are really interchangeable, and that they each enter the proper states for the specified conditions.

• Show that the systems' performance level is maintained when devices are interchanged, or when they fail.

Several types of operations should be performed during configuration test. Some sample operations for testers are

(i) rotate and permutate the positions of devices to ensure physical/logical device permutations work for each device (e.g., if there are two printers A and B, exchange their positions);

(ii) induce malfunctions in each device, to see if the system properly handles the malfunction; (iii) induce multiple device malfunctions to see how the system reacts.

Security testing

Security testing evaluates system characteristics that relate to the availability, integrity, and confidentially of system data and services. Users/clients should be encouraged to make sure their security needs are clearly known at requirements time, so that security issues can be addressed by designers and testers.

Computer software and data can be compromised by: (i) criminals intent on doing damage, stealing data and information, causing denial of service, invading privacy; (ii) errors on the part of honest developers/maintainers who modify, destroy, or compromise data because of misinformation, misunderstandings, and/or lack of knowledge. Both criminal behavior and errors that do damage can be perpetuated by those inside and outside of an organization. Attacks can be random or systematic.

Damage can be done through various means such as:

(i) viruses;

(ii) trojan horses;

(iii) trap doors;

(iv) illicit channels.

The effects of security breaches could be extensive and can cause:

(i) loss of information; (ii) corruption of information; (iii) misinformation; (iv) privacy violations; (v) denial of service.

A password checker can enforce any rules the designers deem necessary to meet security requirements. Password checking and examples of other areas to focus on during security testing are described below.

- Password Checking
- Legal and Illegal Entry with Passwords
- Password Expiration
- Encryption
- Browsing
- Trap Doors
- Viruses

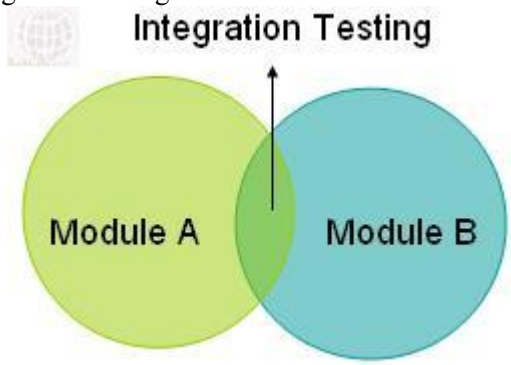
Recovery testing

Recovery testing subjects a system to losses of resources in order to determine if it can recover properly from these losses. This type of testing is especially important for transaction systems, for example, on-line banking software. A test scenario might be to emulate loss of a device during a transaction. Tests would determine if the system could return to a wellknown state, and that no transactions have been compromised. Systems with automated recovery are designed for this purpose. They usually have multiple CPUs and/or multiple instances of devices, and mechanisms to detect the failure of a device. They also have a so-called "checkpoint" system that meticulously records transactions and system states periodically so that these are preserved in case of failure. This

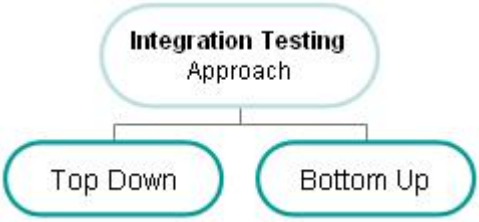
information allows the system to return to a known state after the failure. The recovery testers must ensure that the device monitoring system and the checkpoint software are working properly.

6. Explain in detail about integration testing.

- Integration testing tests integration or interfaces between components, interactions to different parts of the system such as an operating system, file system and hardware or interfaces between systems.
- Also after integrating two different components together we do the integration testing. As displayed in the image below when two different modules 'Module A' and 'Module B' are integrated then the integration testing is done.



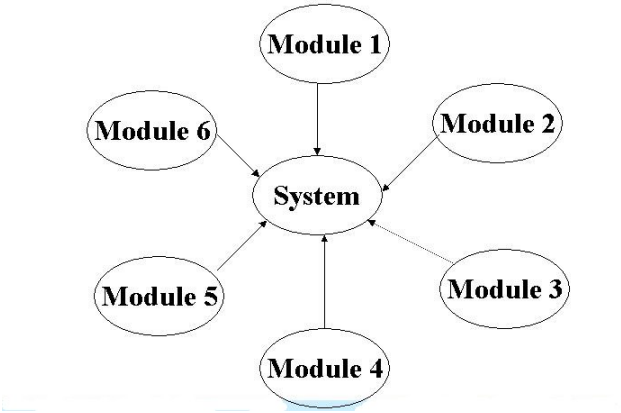
- Integration testing is done by a specific integration tester or test team.
- Integration testing follows two approach known as 'Top Down' approach and 'Bottom Up' approach as shown in the image below:



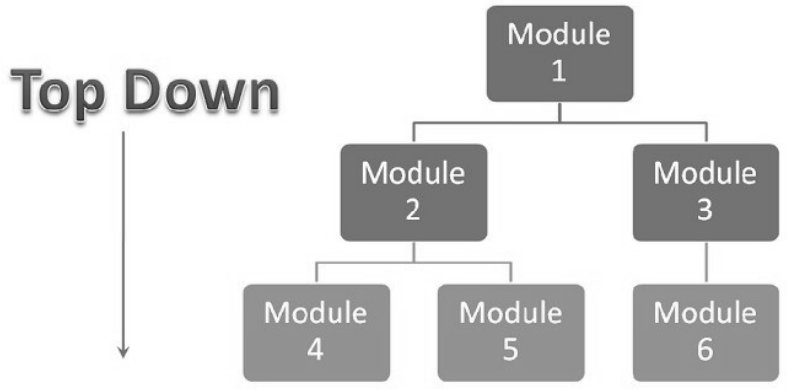
1. Big Bang integration testing:

In Big Bang integration testing all components or modules are integrated simultaneously, after which everything is tested as a whole. As per the below image all the modules from 'Module 1′ to 'Module 6′ are integrated simultaneously then the testing is carried out.

Big Bang Integration Testing



Advantage: Big Bang testing has the advantage that everything is finished before integration testing starts.
Disadvantage: The major disadvantage is that in general it is time consuming and difficult to trace the cause of failures because of this late integration.
2. **Top-down integration testing:** Testing takes place from top to bottom, following the control flow or architectural structure (e.g. starting from the GUI or main menu). Components or systems are substituted by stubs. Below is the diagram of 'Top down Approach':



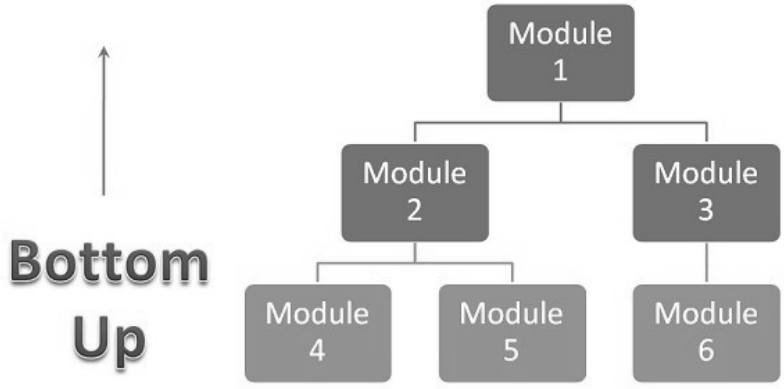
Advantages of Top-Down approach:

- The tested product is very consistent because the integration testing is basically performed in an environment that almost similar to that of reality
- Stubs can be written with lesser time because when compared to the
- drivers then Stubs are simpler to author.

Disadvantages of Top-Down approach:

- Basic functionality is tested at the end of cycle

3. **Bottom-up integration testing:** Testing takes place from the bottom of the control flow upwards. Components or systems are substituted by drivers. Below is the image of 'Bottom up approach':



Advantage of Bottom-Up approach:

- In this approach development and testing can be done together so that the product or application will be efficient and as per the customer
- specifications.

Disadvantages of Bottom-Up approach:

- We can catch the Key interface defects at the end of cycle
- It is required to create the test drivers for modules at all levels except the top control

4. Hybrid Approach:

To overcome the limitations and to exploit the advantages of Top-down and Bottom-up approaches, a hybrid approach in testing is used. As the name suggests, it is a mixture of the two approaches like Top Down approach as well as Bottom Up approach.

In this approach the system is viewed as three layers consisting of the main target layer in the middle, another layer above the target layer, and the last layer below the target layer.

The Top-Down approach is used in the topmost layer and Bottom-Up approach is used in the lowermost layer. The lowermost layer contains many general-purpose utility programs, which are helpful in verifying the correctness during the beginning of testing.

Testing converges for the middle level target layers are selected on the basis of system characteristics and the structure of the code. The middle level target layer contains components using the utilities.

Final decision on selecting an integration approach depends on system characteristics as well as on customer expectations. Sometimes the customer wants to see a working version of the application as soon as possible thereby forcing an integration approach aimed at producing a basic working system in the earlier stages of the testing process.

7. Explain in detail about ad-hoc testing.

When a software testing performed without proper planning and documentation, it is said to be Adhoc Testing. Such kind of tests are executed only once unless we uncover the defects.

Adhoc Tests are done after formal testing is performed on the application. Adhoc methods are the least formal type of testing as it is NOT a structured approach. Hence, defects found using this method are hard to replicate as there are no test cases aligned for those scenarios.
Testing is carried out with the knowledge of the tester about the application and the tester tests randomly without following the specifications/requirements.
Hence the success of Adhoc testing depends upon the capability of the tester, who carries out the test. The tester has to find defects without any proper planning and documentation, solely based on tester's intuition.

Various ways to make Adhoc Testing More Effective

1. **Preparation:** By getting the defect details of a similar application, the probability of finding defects in the application is more.
2. **Creating a Rough Idea:** By creating a rough idea in place the tester will have a focussed approach. It is NOT required to document a detailed plan as what to test and how to test.
3. **Divide and Rule:** By testing the application part by part, we will have a better focus and better understanding of the problems if any.
4. **Targeting Critical Functionalities:** A tester should target those areas that are NOT covered while designing test cases.
5. **Using Tools:** Defects can also be brought to the lime light by using profilers, debuggers and even task monitors. Hence being proficient in using these tools one can uncover several defects.
6. **Documenting the findings:** Though testing is performed randomly, it is better to document the tests if time permits and note down the deviations if any. If defects are found, corresponding test cases are created so that it helps the testers to retest the scenario.

Types of Ad-hoc testing

Buddy testing:

In this form of testing there will be a test member and a development member that will be chosen to work on the same module. Just after the developer completes the unit testing, the tester and developer sit together and work on the module. This kind of testing enables the feature to be viewed in a broader scope for both parties. The developer will gain a perspective of all the different of tests the tester runs and tester will gain a perspective of how the inherent design is which will help him avoid designing invalid scenarios, thereby preventing invalid defects. It will help one think like think the other.

Pair testing:

In this testing, two testers work together on a module with the same test setup shared between them. The idea behind this form of testing to have the two testers brainstorm ideas and methods to have more number of defects. Both can share the work of testing and making necessary documentation of all observations made.

Exploratory testing

Exploratory testing is about exploring, finding out about the software, what it does, what it doesn't do, what works and what doesn't work. The tester is constantly making decisions about what to test next and where to spend the (limited) time. This is an approach that is most useful when there are no or poor specifications and when time is severely limited.

Exploratory testing is a hands-on approach in which testers are involved in minimum planning and maximum test execution.

The planning involves the creation of a test charter, a short declaration of the scope of a short (1 to 2 hour) time-boxed test effort, the objectives and possible approaches to be used.

The test design and test execution activities are performed in parallel typically without formally documenting the test conditions, test cases or test scripts.

Iterative testing

Iterative testing simply means testing that is repeated, or iterated, multiple times. Iterative usability testing matters because the ultimate goal of all usability work is to improve usability, not to catalog problems.

Iterative testing uses the "waterfall model." In Royce's original waterfall model, the following phases are followed perfectly in order:

- Requirements specification and/or problem statement
- Design
- Construction, creation, and writing
- Integration
- Testing, outside reviews, and verification
- Installation
- Maintenance

Agile and Extreme Testing

A software testing practice that follows the principles of agile software development is called Agile Testing. Agile is an iterative development methodology, where requirements evolve through collaboration between the customer and self-organizing teams and agile aligns development with customer needs.

Advantages

- Agile Testing Saves Time and Money
- Less Documentation
- Regular feedback from the end user

- Daily meetings can help to determine the issues well in advance

XP work flow
There are different steps involved in XP methodologies.
- Develop user stories
- Prepare acceptance test cases
- Code
- Test
- Refractor
- delivery

8. Explain in detail about testing Object Oriented Systems.
Testing is a continuous activity during software development. In object-oriented systems, testing encompasses three levels, namely, unit testing, subsystem testing, and system testing.

Unit Testing
In unit testing, the individual classes are tested. It is seen whether the class attributes are implemented as per design and whether the methods and the interfaces are error-free. Unit testing is the responsibility of the application engineer who implements the structure.

Subsystem Testing
This involves testing a particular module or a subsystem and is the responsibility of the subsystem lead. It involves testing the associations within the subsystem as well as the interaction of the subsystem with the outside. Subsystem tests can be used as regression tests for each newly released version of the subsystem.

System Testing
System testing involves testing the system as a whole and is the responsibility of the quality-assurance team. The team often uses system tests as regression tests when assembling new releases.
Object-Oriented Testing Techniques

Grey Box Testing
The different types of test cases that can be designed for testing object-oriented programs are called grey box test cases. Some of the important types of grey box testing are:
- **State model based testing** : This encompasses state coverage, state transition coverage, and state transition path coverage.
- **Use case based testing** : Each scenario in each use case is tested.
- **Class diagram based testing** : Each class, derived class, associations, and aggregations are tested.
- **Sequence diagram based testing** : The methods in the messages in the sequence diagrams are tested.

Techniques for Subsystem Testing
The two main approaches of subsystem testing are:
- **Thread based testing** : All classes that are needed to realize a single use case in a subsystem are integrated and tested.
- **Use based testing** : The interfaces and services of the modules at each level of hierarchy are tested. Testing starts from the individual classes to the small modules comprising of classes, gradually to larger modules, and finally all the major subsystems.

Categories of System Testing

- **Alpha testing** : This is carried out by the testing team within the organization that develops software.
- **Beta testing** : This is carried out by select group of co-operating customers.
- **Acceptance testing** : This is carried out by the customer before accepting the deliverables.

Unit 4
Part A

1. **What is a goal?**
   A goal can be described as (i) a statement of intent, or (ii) a statement of a accomplishment that an individual or an organization wants to achieve.
2. **What are the types of goals?**
   1. Business goal: to increase market share 10% in the next 2 years in the area of financial software.
   2. Technical goal: to reduce defects by 2% per year over the next 3 years.
   3. Business/technical goal: to reduce hotline calls by 5% over the next 2 years.
   4. Political goal: to increase the number of women and minorities in high management positions by 15% in the next 3 years.
3. **Define a policy.**
   A policy can be defined as a high-level statement of principle or course of action that is used to govern a set of activities in an organization.
4. **Define a plan.**
   A plan is a document that provides a framework or approach for achieving a set of goals.
5. **Define a milestone.**
   Milestones are tangible events that are expected to occur at a certain time in the project's lifetime. Managers use them to determine project status.
6. **What are the high-level items included by a planner?**
   The planner usually includes the following essential high-level items.
   - Overall test objectives.
   - What to test (scope of the tests).
   - Who will test.
   - How to test.
   - When to test.
   - When to stop testing.
7. **Give some of the test Plan Components.**
   - Test plan identifier
   - Introduction
   - Items to be tested
   - Features to be tested
   - Approach
8. **What is a Work Breakdown Structure?**
   A Work Breakdown Structure is a hierarchical or treelike representation of all the tasks that are required to complete a project.
9. **What is a cost driver?**
   A cost driver can be described as a process or product factor that has an impact on overall project costs.
10. **What is COCOMO Model?**
    The test planner can use the COCOMO model to estimate total project costs, and then allocate a fraction of those costs for test. Application of the COCOMO model is based on a group of project constants that depend on the nature of the project and items known as cost drivers.
11. **What are Test Design Specification?**
    - Test Design Specification Identifier
    - Features to Be Tested
    - Approach Refinements

- Test Case Identification
- Pass/Fail Criteria

12. What are Test case Specifications?

- Test Case Specification Identifier
- Test Items
- Input Specifications
- Output Specifications
- Special Environmental Needs
- Special Procedural Requirements
- Intercase Dependencies

13. What are Test Summary Report?

- Test Summary Report identifier
- Variances
- Comprehensiveness assessment
- Summary of results
- Evaluation
- Summary of activities
- Approvals

14. What are the responsibilities for the developers/testers?

- Working with management to develop testing and debugging policies and goals.
- Participating in the teams that oversee policy compliance and change management.
- Familiarizing themselves with the approved set of testing/debugging goals and policies, keeping up-to-date with revisions, and making suggestions for changes when appropriate.
- When developing test plans, setting testing goals for each project at each level of test that reflect organizational testing goals and policies.
- Carrying out testing activities that are in compliance with organizational policies.

15. List the Personal and Managerial Skills.

- Organizational, and planning skills
- Track and pay attention to detail
- Determination to discover and solve problems
- Work with others, resolve conflicts
- Mentor and train others
- Work with users/clients
- Written/oral communication skills
- Think creatively

16. List some of the Technical Skills.

- General software engineering principles and practices
- Understanding of testing principles and practices
- Understanding of basic testing strategies, and methods
- Ability to plan, design, and execute test cases
- Knowledge of process issues

17. What is the role of Test Lead?

The test lead assists the test manager and works with a team of test engineers on individual projects. He or she may be responsible for duties such as test planning, staff supervision, and status reporting. The test lead also participates in test design, test execution and reporting, technical reviews, customer interaction, and tool training.

18. What is the role of Test Engineer?

The test engineers design, develop, and execute tests, develop test harnesses, and set up test laboratories and environments. They also give input to test planning and support maintenance of the test and defect repositories.

19. What is the role of the Junior Test Engineer?

The junior test engineers are usually new hires. They gain experience by participating in test design, test execution, and test harness development. They may also be asked to review user manuals and user help facilities defect and maintain the test and defect repositories.

20. Give some of the knowledge that must be shown by Candidates for Certification.

Candidates for certification must show knowledge in areas that include:

- Quality management;
- Project management;
- Measurement;
- Testing;
- Audits;
- Configuration management.

21. What is the use of V-model in testing?

The V-model is the model that illustrates how the testing activities can be integrated in to each phase of the standard software life cycle.

22. Explain the test team hierarchy.

- The test manager
- The test lead
- The test engineer
- The junior test engineer

23. What are the steps in forming the test group?

- Upper management support for test function
- Establish test group organization
- Define education and skill levels
- Develop job description
- Interview candidates
- Select test group members

24. Define the term pass/Fail Criteria.

Given a test item and a test case, the tester must have a set of criteria to decide on whether the test has been passed or failed upon execution.

25. Define suspension and resumption criteria.

The criteria to suspend and resume testing are described in the simplest of cases testing is suspended at the end of a working day and resumed the following morning.

Part B

1. Briefly explain about Organization structure for testing teams.

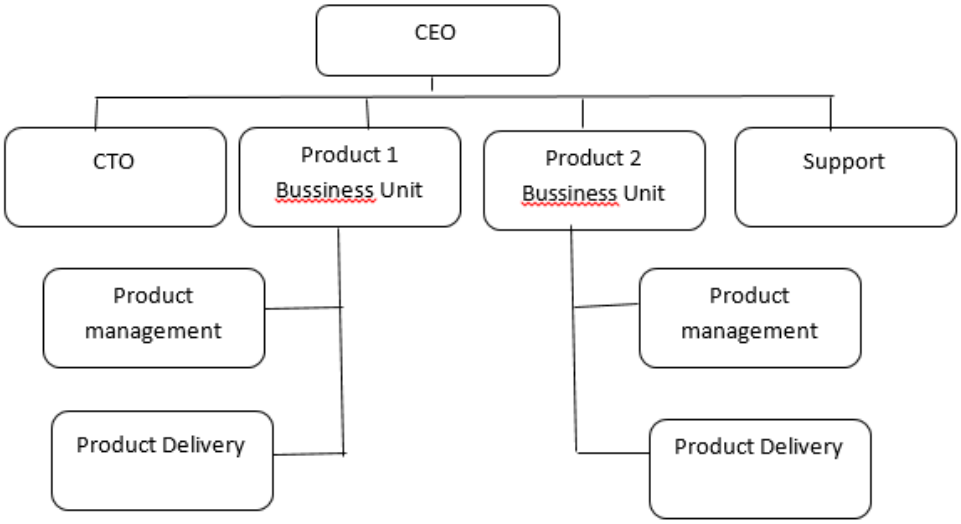
Organization structures directly relate to some of the people issues.

In addition, the study of organization structures is important from the point of view of effectiveness because an appropriately designed organization structure can provide accountability to results.

This accountability can promote better teamwork among the different constituents and create in better focus in the work.

In addition, organization structures provide a road map for the team members to envision their career paths.

Structures for Multi-Product Companies:
When a company becomes successful as a single-product company, it may decide to diversify into other products. In such a case, each of the products is considered as a separate business unit, responsible for all activities of a product. In addition, as before, there will be common roles like the CTO.



Effects of Globalization and Geographically Distribute Teams on Product Testing:

Business Impact of Globalization:
Globalization has revolutionized the way we produce and maintain software products.
1. Markets for software products are becoming global. Hence, a global distribution of production of software becomes necessary to exploit the knowledge of the local conditions.
2. Since the markets are global, the needs that a product must satisfy are increasing exponentially. Hence, it is impossible to meet all the demands from resources from just one location.
3. Several countries around the globe have a rich supply of talented people and these needs to be utilized effectively.
4. Some countries offer not only a rich supply of talent pool but also offer cost advantages that makes them a compelling business proposition.

2. **Explain about Test plan components in detail.**
1. Test Plan Identifier:
Each test plan should have a unique identifier so that it can be associated with a specific project and become a part of the project history. The project history and all project-related items should be stored in a project database or come under the control of a configuration management system.
2. Introduction

The test planner gives an overall description of the project, the software system being developed or maintained, and the software items and or features to be tested. It is useful to include a high-level description of testing goals and the testing approaches to be used.

3. Items to Be Tested

This is a listing of the entities to be tested and should include names, identifiers, and version/revision numbers for each entity. The items listed could include procedures, classes, modules, libraries, subsystems, and systems.

4. Features to Be Tested:

In this component of the test plan the tester gives another view of the entities to be tested by describing them in terms of the features they encompass.

5. Approach

The planner should also include for each feature or combination of features, the approach that will be taken to ensure that each is adequately tested. Tools and techniques necessary for the tests should be included. Expectations for test completeness and how the degree of completeness will be determined should be described.

6. Item Pass/Fail Criteria

Given a test item and a test case, the tester must have a set of criteria to decide on whether the test has been passed or failed upon execution. The master test plan should provide a general description of these criteria. In the test design specification section more specific details are given for each item or group of items under test with that specification

7. Suspension and Resumption Criteria

Testing is suspended at the end of a working day and resumed the following morning.

For some test items this condition may not apply and additional details need to be provided by the test planner.

The test plan should also specify conditions to suspend testing based on the effects or criticality level of the failures/defects observed.

8. Test Deliverables

Execution-based testing has a set of deliverables that includes the test plan along with its associated test design specifications, test procedures, and test cases

9. Testing Tasks

Using a Work Breakdown Structure (WBS) is useful here**.** A Work Breakdown Structure is a hierarchical or treelike representation of all the tasks that are required to complete a project.

10. The Testing Environment

Here the test planner describes the software and hardware needs for the testing effort. For example, any special equipment or hardware needed such as emulators, telecommunication equipment, or other devices should be noted.

11. Responsibilities

The staff who will be responsible for test-related tasks should be identified.

This includes personnel who will be:

- transmitting the software-under-test;
- developing test design specifications, and test cases;
- executing the tests and recording results;
- tracking and monitoring the test efforts;
- checking results;
- interacting with developers;
- managing and providing equipment;
- developing the test harnesses;
- interacting with the users/customers.

12. Scheduling

Task durations should be established and recorded with the aid of a task networking tool. Test milestones should be established, recorded, and Scheduled. These milestones usually appear in the project plan as well as the test plan. They are necessary for tracking testing efforts to ensure that Actual testing is proceeding as planned.

13. Risks and Contingencies

Every testing effort has risks associated with it. Testing software with a high degree of criticality, complexity, or a tight delivery deadline all impose risks that may have negative impacts on project goals.

These risks should be

(i) identified,

(ii) evaluated in terms of their probability of occurrence,

(iii) prioritized, and

(iv) contingency plans should be developed that can be activated if the risk occurs.

14. Testing Costs

The IEEE standard for test plan documentation does not include a separate cost component in its specification of a test plan.

Test costs that should include in the plan are:

- Costs of planning and designing the tests;
- Costs of acquiring the hardware and software necessary for the tests
- Costs of executing the tests;
- Costs of recording and analyzing test results;
- Tear-down costs to restore the environment.

Test planners often borrow cost estimation techniques and models from project planners and apply them to testing.

3. Discuss on test plan attachments.

The reader may be puzzled as to where in the test plan are the details needed for organizing and executing the tests.

For example, what are the required inputs, outputs, and procedural steps for each test; where will the tests be stored for each item or feature; will it be tested using a black box, white box, or functional approach?

The following components of the test plan contain this detailed information. These documents are generally attached to the test plan.

Requirement Identifier	Requirement Description	Priority	Status	Test ID
SR-25-13.5	Displays opening screens	8	Yes	TC-25-2 TC-25-5
SR-25-52.2	Checks the validity of user password	9	Yes	TC-25-18 TC-25-23

Test Design Specification:

It is used to identity the features covered by this design and associated tests for the features.

The test design specification also has links to the associated test cases and test procedures needed to test the features, and also describes in detail pass/fail criteria for the features.

The test design specification helps to organize the tests and provides the connection to the actual test inputs/outputs and test steps.

Test design specification should have the following components according to the IEEE standard

Test Design Specification Identifier
Give each test design specification a unique identifier and a reference to its associated test plan.

Features to Be Tested
Test items, features, and combination of features covered by this test design specification are listed. References to the items in the requirements and/or design document should be included.

Approach Refinements
In the test plan a general description of the approach to be used to test each item was described. In this document the necessary details are added.

For example, the specific test techniques to be used to generate test cases are described, and the rational is given for the choices. The test planner also describes how test results will be nalyzed. For example, will an automated comparator be used to compare actual and expected results?

Test Case Identification
Each test design specification is associated with a set of test cases and a set of set procedures. The test cases contain input/output information, and the test procedures contain the steps necessary to execute the tests.
A test case may be associated with more than one test design specification.

Pass/Fail Criteria
In this section the specific criteria to be used for determining whether the item has passed/failed a test is given.

Test Procedure Specification:
Procedure Steps

(i) Setup: to prepare for execution of the procedure;
(ii) Start: to begin execution of the procedure;
(iii) Proceed: to continue the execution of the procedure;
(iv) Measure: to describe how test measurements related to outputs will be made;
(v) Shut down: to describe actions needed to suspend the test when unexpected events occur;
(vi) Restart: to describe restart points and actions needed to restart the procedure from these points;
(vii) Stop: to describe actions needed to bring the procedure to an orderly halt;
(viii) Wrap up: to describe actions necessary to restore the environment;
(ix) Contingencies: plans for handling anomalous events if they occur during execution of this procedure.

4. **How will you report the test result? Explain in detail.**
The test plan and its attachments are test-related documents that are prepared prior to test execution.
There are additional documents related to testing that are prepared during and after execution of the tests.
The IEEE Standard for Software Test Documentation describes the following documents
Test Log

The test log should be prepared by the person executing the tests. It is a diary of the events that take place during the test.

Test Log Identifier

Each test log should have a unique identifier

- Description,
- Activity and Event Entries,
- Execution description,
- Procedure results,
- Environmental information,
- Anomalous events,
- Incident report identifiers.

Test Incident Report

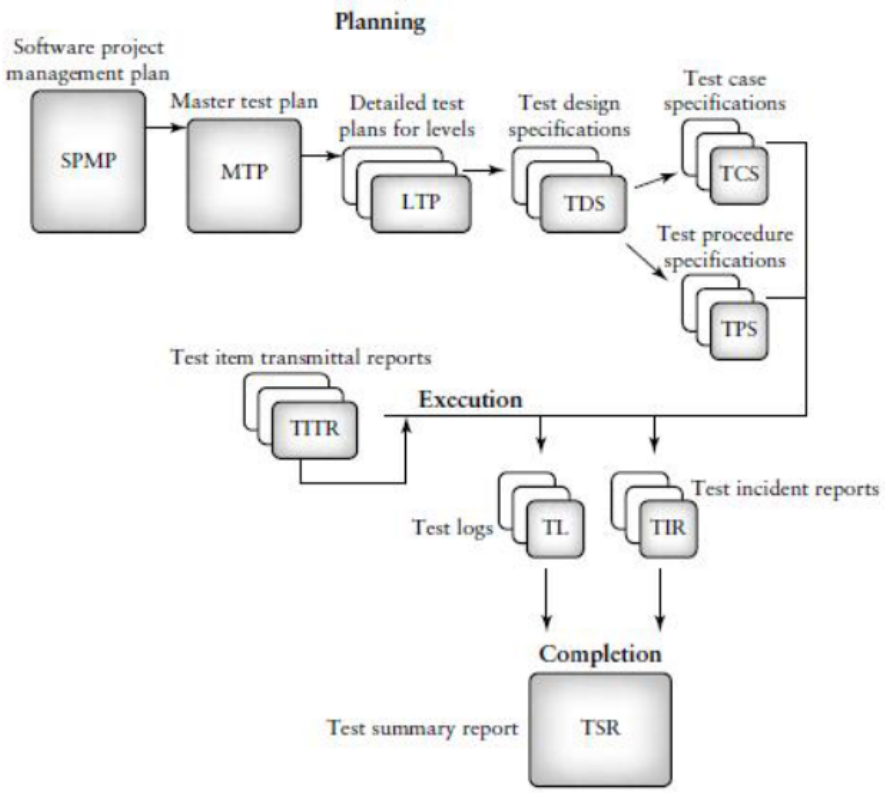
The IEEE Standard for Software Test Documentation recommends the following sections in the report

1. Test Incident Report identifier
2. Summary
3. Incident
4. Impact

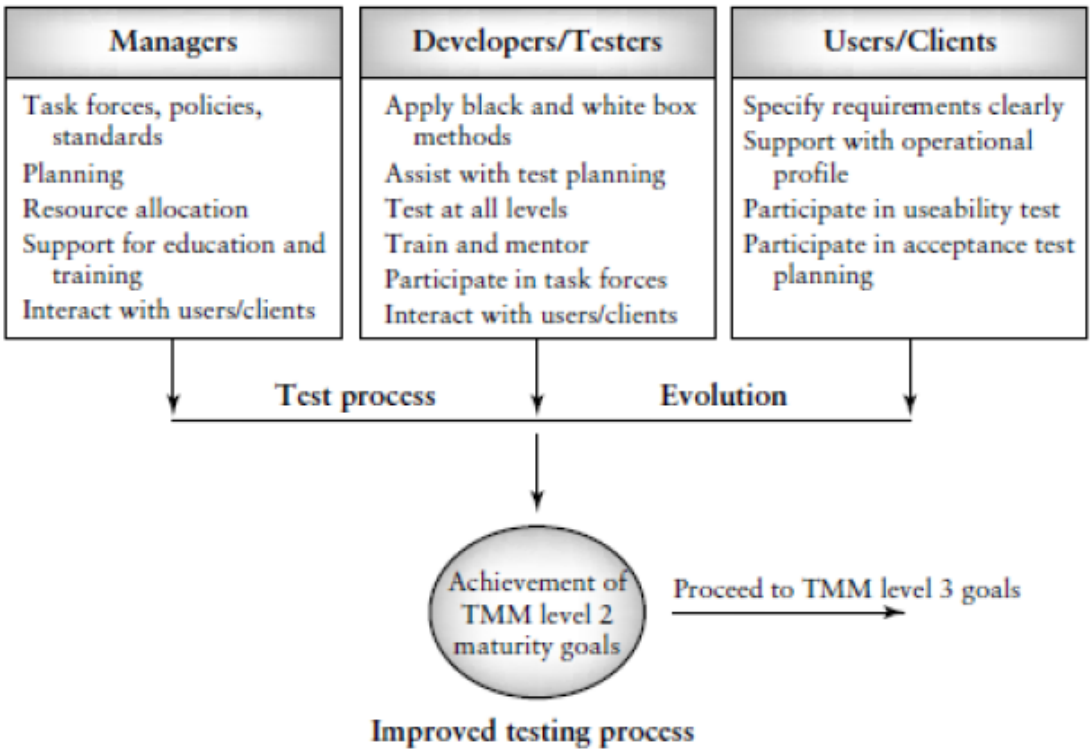
Test Summary Report

This report is prepared when testing is complete. It is a summary of the results of the testing efforts. It also becomes a part of the project‘s historical database and provides a basis for lessons learned as applied to future projects

- Test Summary Report identifier Variances:
- Comprehensiveness assessment
- Summary of results
- Evaluation
- Summary of activities
- Approvals



The Role of the Three Critical Groups in Testing Planning and Test Policy Development
For the TMM maturity goal, Develop Testing and Debugging Goals



The TMM recommends that *project* and *upper management:*

- Provide access to existing organizational goal/policy statements and sample testing policies
- Provide adequate resources and funding to form the committees (team or task force) on testing and debugging. Committee makeup is managerial, with technical staff serving as co members.
- Support the recommendations and policies of the committee by: distributing testing/debugging goal/policy documents to project managers, developers, and other interested staff, appointing a permanent team to oversee compliance and policy change making.
- Ensure that the necessary training, education, and tools to carry out defined testing/debugging goals is made available.
- Assign responsibilities for testing and debugging.

5. **Discuss on the different skills needed by the test specialist.**

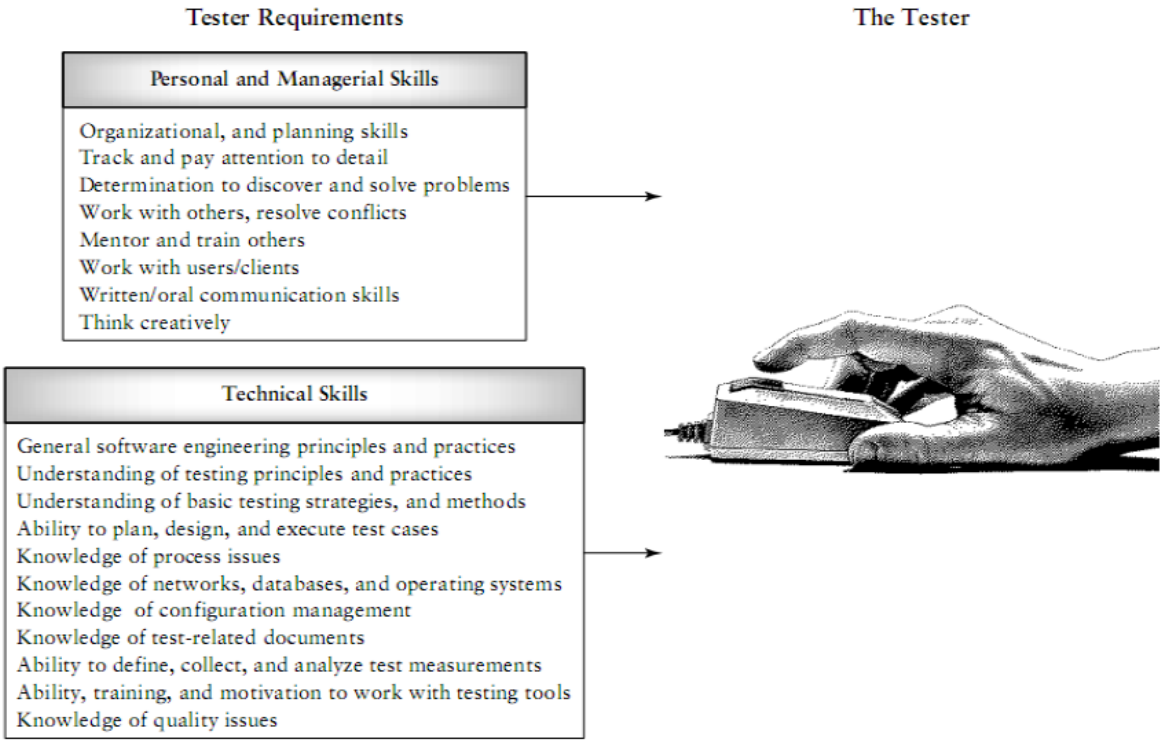
Managerial and personal skills are necessary for success in the area of work. On the personal and Managerial level a test specialist must have:

- Organizational, and planning skills;
- The ability to keep track of, and pay attention to, details;
- The determination to discover and solve problems;
- The ability to work with others and be able to resolve conflicts;
- The ability to mentor and train others;
- The ability to work with users and clients;

- Strong written and oral communication skills;

On the *technical* level testers need to have:

- an education that includes an understanding of general software engineering principles, practices, and methodologies;
- strong coding skills and an understanding of code structure and behavior;
- a good understanding of testing principles and practices;
- a good understanding of basic testing strategies, methods, and techniques;
- the ability and experience to plan, design, and execute test cases and test procedures on multiple levels (unit, integration, etc.);
- a knowledge of process issues;
- knowledge of how networks, databases, and operating systems are
- organized and how they work;
- a knowledge of configuration management;
- a knowledge of test-related documents and the role each documents
- plays in the testing process;
- the ability to define, collect, and analyze test-related measurements;
- the ability, training, and motivation to work with testing tools and equipment;
- a knowledge of quality issues.



6. Explain about people and organizational issues in testing.

Common People Issues:

Perceptions and misconceptions about testing*:*

Look at "heardin the street" statements about the testing profession. These statements are sometimes made by testing professionals, by the management team, and by academicians.

Where these statements come from, thefallacy in the statements, and what can be given as arguments to counter these perceptions.

"Testing is not Technically Challenging"

If you are conducting interviews to hire people for performing testing functions, you will generally observe a very bipolar behavior among the candidates.
There will be the first set of people—usually a **minority**—who will approach testing with tremendous pride, commitment, and enjoyment.

The second set—unfortunately, a **majority**—will be those who get into testing, ―because they have no choice.

Functions in development	Correspondinf functions in testing projects	Similarities
Requirement specification	Test specification	Both requires a through understanding of the domain
Design	Test Design	Test design carries with it all attributes of product design in terms of architecting the test system
Development/coding	Test script development	Involves the using of test development and test automation tool
Testing	Making tests operational	This would involve well-knit teamwork between the development and testing team to ensure that the correct result are captured
Maintenence	Test maintenance	Keep the test current with changes from maintenance

Comparison between Testing and Development Functions:

- Testing is often a crunch time function.
- Generally more ―elasticity‖ is allowed in projects in earlier phases.
- Testing functions are arguably the most difficult ones to staff.
- Testing functions usually carry more external dependencies than development functions.

Providing Career Paths For Testing Professionals:
When people look for a career path in testing (or for that matter in any chosen profession), some of the areas of progression they look for are

- Technical Challenge.
- Learning opportunities.
- Increasing responsibility and authority.
- Increasing Independence.
- Organizations Success.
- Rewards and Recognition.

The Role of the Ecosystem and a Call for Action:
The perceptions, misconceptions, and issues discussed so far cannot all be corrected by each and every organization individually.
There are collective and much higher-level actions that need to be done.
These actions pertain to the entire ecosystem covering the education system, senior management, and the community as a whole.

Role of Education System:
The education system does not place sufficient emphasis on testing. Consider some of these facts about what prevails in most universities.

7. **Brief on test planning concepts.**
A plan is a document that provides a framework or approach for achieving a set of goals. In the software domain, plans can be strictly business oriented,

For example, Long-term plans to support the economic growth of an organization, or they can be more technical in nature, for example, a plan to develop a specific software product.

Test planning is an essential practice for any organization that wishes

To develop a test process that is repeatable and manageable. Pursuing the Maturity goals embedded in the TMM structure is not a necessary precondition for initiating a test-planning process.

However, a test process Improvement effort does provide a good framework for adopting this essential practice.

Test planning should begin early in the software life Cycle; **Milestones** are tangible events that are expected to occur at a certain time in the project‘s lifetime. Managers use them to determine project status.

The planner usually includes the following essential high-level items.
1. *Overall test objectives.* As testers, why are we testing, what is to be achieved by the tests, and what are the risks associated with testing this product?

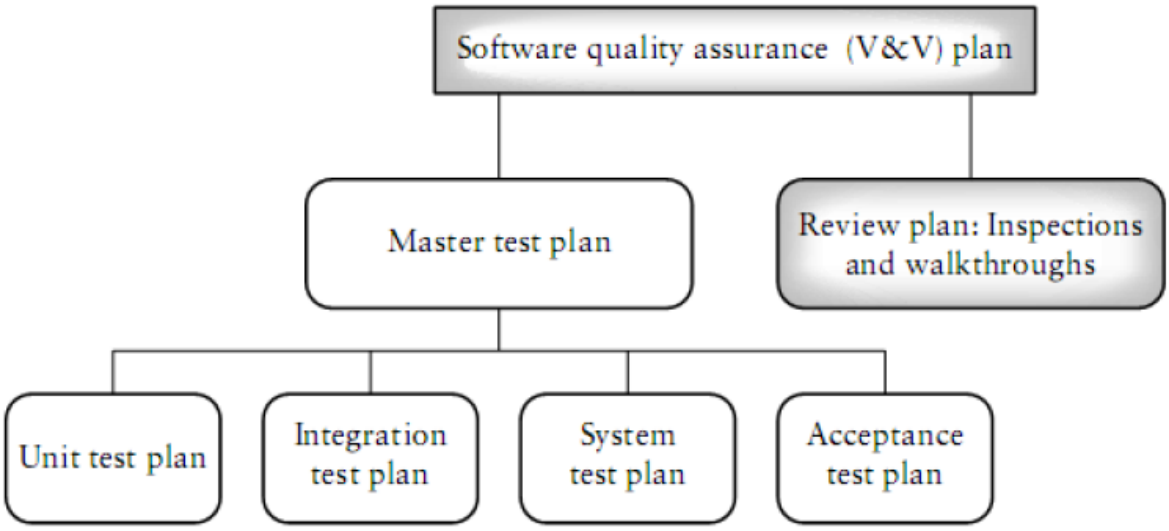
2. *What to test (scope of the tests).* What items, features, procedures, functions, objects, clusters, and subsystems will be tested?

3. *Who will test.* Who are the personnel responsible for the tests?

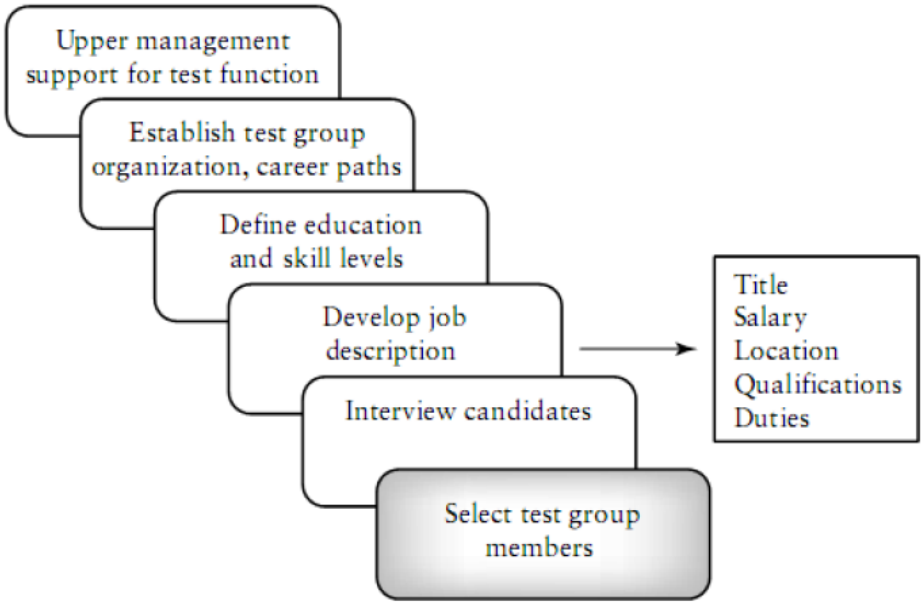
4. *How to test.* What strategies, methods, hardware, software tools, and techniques are going to be applied? What test documents and deliverable should be produced?

5. *When to test.* What are the schedules for tests? What items need to be available?

6. *When to stop testing.* It is not economically feasible or practical to plan to test until all defects have been revealed. This is a goal that testers can never be sure they have reached. Because of budgets, scheduling, and customer deadlines, specific conditions must be outlined in the test plan that allow testers/managers to decide when testing is considered to be complete.



8. **How will you build a test group? Explain it.**
Staffing activities include filling positions, assimilating new personnel, education and training, and staff evaluation.



To initiate the process, upper management must support the decision to establisha test group and commit resources to the group. Decisions must be made on how the testing group will be organized, what career paths are available, and how the group fits into the organizational structure.

When hiring staff to fill test specialist positions, management should have a clear idea of the educational and skill levels required for each testing position and develop formal job descriptions to fill the test group slots.

Dustin describes a typical job requisition for a test specialist. Included on this requisition are the job title, full time/part time, location, salary, location, qualifications that are required (the applicant must have these), qualifications that are desired (the recruiter is flexible on these), and a description of the duties.

When the job description has been approved and distributed, the interviewing process takes place. Interviews should be structured and of a problem-solving nature.

The interviewer should prepare an extensive list of questions to determine the interviewee's technical background as well as his or her personal skills and motivation.

Zawacki has developed a general guide for selecting technical staff members that can be used by test managers .
Dustin describes the kinds of questions that an interviewer should ask when selecting a test specialist.

When the team has been selected and is up and working on projects, the team manager is responsible for keeping the test team positions filled (there are always attrition problems). He must continually evaluate team member performance.

Bartol and Martin have written a paper that contains guidelines for evaluation of employees that can be applied to any type of team and organization .

They describe four categories for employees based on their performance:
(i) retain,
(ii) likely to retain,
(iii) likely to release,
(iv) and release.

Unit 5
Part A

1. **What is Test Automation?**
   In software **testing**, **test automation** is the use of special software (separate from the software being tested) to control the execution of **tests** and the comparison of actual outcomes with predicted outcomes.
   - Automation saves times as software can execute test cases faster than human do
   - Test automation can free the test engineers from mundane tasks and make focus on creative task.
   - Automated tests can be more reliable.
   - Automation helps in immediate testing
   - Automation protect an organization against attrition of test engineers
   - Test automation opens up best opportunities for better utilization for global resources
   - Certain types of testing cannot be executed without automation.
   - Automation means end to end, not test execution alone.
2. **Define test data generator.**
   Automation should have scripts that produce test data to maximize coverage of permutations and combinations of inputs and expected output or result comparison. They are called test data generator.
3. **What are the different generations of automation?**
   - First Generation – Record and playback
   - Second Generation – Data Driven

- Third Generation – Action driven

4. **What are the scopes for automation?**
   - Identifying the types of testing amenable to automation
   - Automating areas less prone to changes
   - Automate test that pertain to standards.
   - Management aspects in automation.
5. **What is configuration file?**
   - A configuration file contains a set of variables that are used in automation.
   - The variables could be for the test framework or for other modules in automation such as tools and metrics or for the test suite or for a set of test cases or for a particular test case
   - A configuration file is important for running the test cases for various execution conditions and for running test for various input and output conditions.
   - The values of this variables in this configuration file can be changed dynamically to achieve different execution input, output and state condition.
6. **Define test framework.**
   A testing framework or more specifically a testing automation framework is an execution environment for automated tests. It is the overall system in which the tests will be automated.
   It is defined as the set of assumptions, concepts, and practices that constitute a work platform or support for automated testing.
   The Testing framework is responsible for:
   - Defining the format in which to express expectations.
   - Creating a mechanism to hook into or drive the application under test
   - Executing the tests
   - Reporting results
7. **Explain the work of report generator.**
   - Once the test run result is available, the next step is to prepare the test report and metrics. Preparing report is a complex and time consuming effort and hence it should be part of the automation design.
   - The module that takes input and prepares a formatted report is a called report generator.
   - The periodicity of the report is different such as daily, weekly, monthly or milestones report.
8. **What are the criteria for selecting test tools?**
   - Meeting requirements
   - Technology expectation
   - Training/skills
   - Management aspects
9. **Define instrumented code.**
   Test tool requires their libraries to be linked with product binaries. When the libraries are linked with the source code of the product, it is called instruments code.
10. **List down the basic concepts of extreme programming.**
    - Unit test cases are developed before the coding phase starts.
    - Code is written for test cases and are written to ensure test cases pass
    - All the unit tests must run 100% all the time
    - Everyone owns the product; they often cross boundaries.
11. **What is metrics program and list down the steps in metrics program?**
    Metris drive information from raw data with the view to help in decision making.
    Steps:
    - Identify what to measure

- Transform measurement to metrics
- Decide operational requirements
- Perform metrics analysis
- Take actions and follow up
- Refine

12. List out the different types of metrics.

- Project metrics: A set of metrics that indicate how the project is planned and executed.
- Progress metrics: A set of metrics that tracks how the different activities of the project are progressing
- Productivity metrics: A set of metrics that helps in planning and estimating of testing activities.

13. Define effort variance and schedule variance.

Effort variance provides a quantitative measure of the relative difference between the revised and actual efforts.

Schedule variance is the deviation of the actual schedule from the estimated schedule.

14. What is test defect metrics?

- A set of metrics helps to understand how the defects that are found can be used to improve testing and product quality.
- The defects are classified by defect priority and defect severity.
- Defect priority provides management perspective for the order of defect fixes.
- The severity of defects provides the test team a perspective of the impact of that defect in product functionality.

15. What do you mean by closed defect distribution?

- The objective of testing is not only to find defects.
- The testing team also has the objective to ensure that all defects fond through testing are fixed so that the customer gets the benefit of testing and the product quality improves.
- To ensure the most of the defects are fixed, the testing team has to track the defects and analyze how they are closed. The closed defect distribution helps in this analysis.

16. Define release metrics.

- The decision to release a product would need to consider several perspectives and several metrics. The release metrics provides the guidelines in making the release decision.
- Some release metrics are test case executed, effort distribution, defect find rate, defect fix rate, outstanding defect trends, priority outstanding defect trends, weighted defect trends, defect density and defect removal rate, age analysis of outstanding defects, introduces and reopened defects, defects per100 hours of testing, test cases executed for 100 hours of testing, test phase effectiveness, closed defect distribution.

17. What is the different purpose of productivity metrics?

- Estimating for the new release
- Finding out how well the team is progressing, understanding the reason for variation in result.
- Estimating the number of defects that can be found
- Estimating release date and quality
- Estimating the cost involved in the release

18. What do you mean by defect per 100 Hours of testing?

Defect per 100 hours of testing normalizes the number of defects found in the product with respect to the effort spend

Defect per 100 hours of testing=(Total defect found in the product for a period/Total hours spend to get those defects)*100

19. **How the defect metrics can be used in improving the development activities?**
    - Component wise defect distribution
    - Defect density and defect removal rate
    - Age analysis of outstanding defects
    - Introduced
20. **What are the different test defect metrics?**
    - Defect find rate
    - Defect fix rate
    - Outstanding defects rate
    - Priority outstanding rate
    - Weighted defects trend
    - Defect cause distribution
21. **List various measurements for monitoring the testing status.**
    Coverage measures, Test case development, Test execution, Test harness development
22. **What are the various severity level hierarchy?**
    - Catastrophic, Critical, Marginal, Minor or annoying
23. **What are the disadvantages of first generation automation?**
    - Scripts holds hardcoded values
    - Test maintenance cost is maximized
24. **What are the types of reports?**
    - Executive report: gives a very level status
    - Technical report: moderate level of details of test run
    - Detailed or debug report: to debug the failed test case
25. **What are the different software automation testing tools?**
    - Rational functional tester
    - Robot framework
    - Sahi
    - Selenium

Part B

1. **a. Explain the framework for test automation.**

   A test case is asset of sequential steps to execute a test operating on a set of predefined inputs to produce certain expected outputs.
   There are two types of test cases
   - Automated: An automated test case is executed using automation
   - Manual: A manual test case is executed manually

   A test case can be represented in many forms. It can be documented as a set of simple steps, or an assertion or a set of assertions.

   Testing involves several phases and several types of testing. Some test cases are repeates several times during a product release because the product is built several times.

   Not only the test cases repetitive in testing, some operations in the test cases too are repetitive.

   For eg; “Log in to the system” ar performed in a large number of test cases for a product.

This presents an opportunity for the automation code to be reused for different purposes and scenarios

Log in can be tested for different types of testing

S.No	Test cases for testing	Belongs to what type of testing
1	Check in whether log in works	Functionality
2	Repeat log in operation in a loop for 48 hours	Reliability
3	Perform log in from 10000 clients	Load/stress testing
4	Measure time taken for log in operations in different conditions	Performance
5	Run log in operation from a machine running Japanese language	Internationalization

There are two important dimensions: "What operations have to be tested" and "How the operations have to be tested" The how portion of test case is called scenarios. "what an operation has to do" is a product specific feature and "how they are to be run" is a framework specific requirement

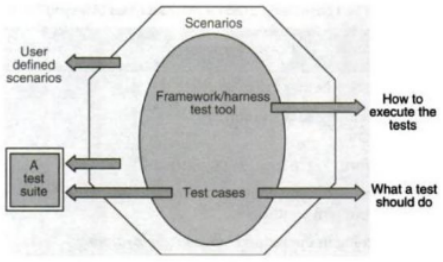
The automation is based on the fact that product operation are repetitive in nature and by automating the basic operations and leaving the different scenarios to the framework tool, great progress can be made.

This ensures code reuse for automation and draws a clear boundary between "what a test suite has to do" and "what a framework or a test tool should complement".

When scenario are combined by basic operations of the product, they become automated test cases.

When a set of test cases is combined and associated with a set of scenarios, they are called test suites.

A test suite is nothing but a set of test cases that are automated and scenarios that are associated with the test cases.



b. Brief the "Generation of Automation".

The skills required depends on what generation of automation the company is in.

1) Record / playback and test harness tools (first generation).
2) Data driven tools (second generation).
3) Action driven (third generation).

1) Record / Playback and Test Harness Tools:

- One of the most boring and time-consuming activity during testing life cycle is to rerun manual tests number of times.
- Here, record/playback tools are of great help to the testers. These tools do this by recording and replaying the test input scripts.
- As a result, tests can be replayed without attendant for long hours specially during regression testing.
- Also these recorded test scripts can be edited as per need i.e., whenever changes are made to the software.
- These tools can even capture human operations e.g., mouse activity, keystrokes etc.
- A record / playback tool can be either intrusive or non-intrusive. Intrusive record / playback tools are also called native tools as they along with software-under-test (SUT), reside on the same machine.
- Non-intrusive record / playback tools on the other hand, reside on the separate machine and is connected to the machine containing software to be tested using special hardware.
- One advantage of these tools is to capture errors which users frequently make and which developers cannot reproduce.
- Test harness tools are the category of record / playback tools used for capturing and replaying sequence of tests.
- These tools enable running large volume of tests unattended and help in generating reports by using comparators.
- These tools are very important at CMM level - 2 and above.

2) Data-driven Tools:

- This method help in developing test scripts that generates the set of input conditions and corresponding expected output. The approach takes as much time and effort as the product.
- However, changes to application do not require the automated test cases to be changed as long as the input conditions and expected output are still valid.
- This generation of automation focuses on input and output conditions using the black box testing approach.

3) Action-driven Tools:

- This technique enables a layman to create automated tests.
- There are no input and expected output conditions required for running the tests.
- All actions that appear on the application are automatically tested, based on a generic set of controls defined for automation.
- From the above approaches / generations of automation, it is clear that different levels of skills are needed based on the generation of automation selected.

Automation – First generation	Automation – second generation	Automation – Third generation	
Skills for test case automation	Skills for test case automation	Skills for test case automation	Skills for framework
Scripting languages	Scripting languages	Scripting languages	Programming languages
Record – playback tool usage	Programming languages	Programming languages	Design and architecture skills for framework creation
	Knowledge of data generation techniques	Design and architecture of the product under test	Generic test requirements for multiple products
	Usage of the product under test	Usage of the framework	

2. a. Discuss the scope of automation in detail.

The automation requirements define what needs to be automated looking into various aspects.

The specific requirement can vary from product to product, from situation to situation, from time to time.

Identifying the types of testing amenable to automation

Certain types of tests automatically lend themselves to automation.

Stress, reliability, scalability and performance testing
These types of testing require the test cases to be run from large number of different machines for an extended period of time, such as 24 hours, 48 hours, and so on. It is just not possible to have hundreds of users trying out the product day in and day out-they may neither be willing to perform the repetitive tasks, nor will it be possible to find that many people with the required skill sets. Test cases belonging to these testing types become the first candidates for automation.

Regression Testing
Regression tests are repetitive in nature. These test cases are executed multiple times during the product development phases. Given the repetitive nature of the test cases, automation will save significant time and effort in the long run.

Functional testing
These kind of tests may require a complex set up and thus require specialized skill, which may not be available on an ongoing basis. Automating these once, using the expert skill sets, can enable using less-skilled people to run these tests on an ongoing basis. As a thumb rule, if

test cases need to be executed at least ten times in the near future, say , one year, and if the effort for automation does not exceed ten times of executing those test cases, then they become candidates for automation.

Automating Areas Less Prone to Change

Automation should consider those areas where requirements go through lesser or no changes. Normally change in requirements cause scenarios and new features to be impacted, not the basic functionality of the product. While automating, basic functionality of the product has to be considered first, so that they can be used for "regression test bed" and "daily builds and smoke test."

Automate Tests that Pertain to Standards

Automating for standards provides a dual advantage. Test suites developed for standards are not only used for product testing but can also be sold as test tools for the market. A large number of tools available in the commercial market were internally developed for in house usage.

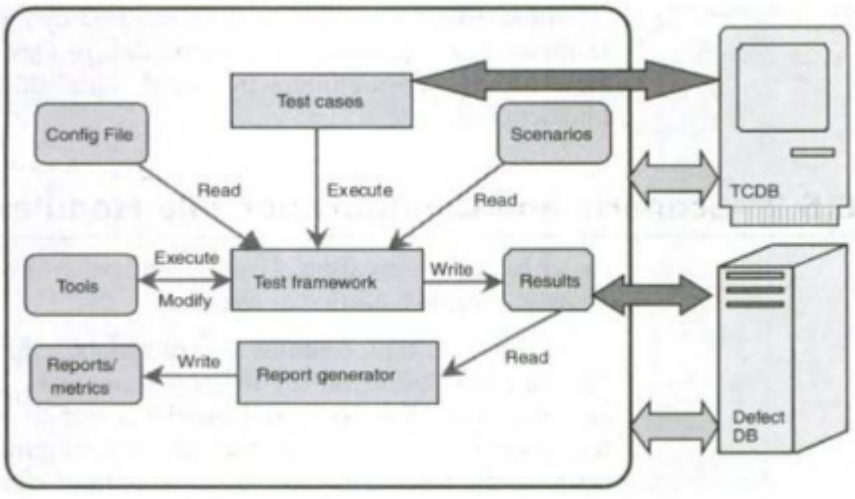
Hence, automating for standards creates new opportunities for them to be sold as commercial tools.

In case there are tools already available in the market for checking such standards, then there is no point in reinventing the wheel and rebuilding these tests. Rather, focus should be towards other areas for which tools are not available and in providing interfaces to other tools.

Testing for standards have certain legal and organizational requirements. To certify the software or hardware, a test suite is developed and handed over to different companies. The certification suites are executed every time by the supporting organization before the release of software and hardware.

This is called "certification testing" and requires perfectly compliant results every time the tests are executed. The companies that do certification testing may not know much about the product and standards but do the majority of this testing. Hence, automation in this area will go a long way. This is definitely an area of focus for automation.

b. Explain the design and architecture for automation.



Architecture for test automation involves two major heads: a test infrastructure that covers a test case database and a defect database or defect repository. Using this infrastructure, test framework provides a backbone that ties the selection and execution of test cases.

External modules:

There are two modules that are external modules to automation – TCDB and defect DB. All test cases, the steps to execute them, and the history of their execution are stored in TCDB. The test cases in the TCDB can be manual or automated. The manual test cases do not need any interaction between the framework and TCDB.

Defect DB or defect database contains details of all the defects that are found in various products that re tested in a particular organization. It contains defects and all the related information. Test engineers submit the defects for manual test cases. For automated test cases, the framework can automatically submit the defects to the defect DB during execution.

Scenario and Configuration File Modules:

A configuration file contains a set of variables that are used in automation. The variables could be for the test framework or for other modules in automation such as tools and metrics or for the test suite or for a set of test cases or for a particular test case. A configuration file is important for running the test cases of various execution conditions and for running the tests for various input and output conditions and states.

Test cases and Test framework Modules:

Test case is an object for execution for the other modules in the architecture and does not represent any interaction by itself. A test framework is a module that combines "what to execute" and "how they have to be executed". It picks up the specific test cases that are automated from TCDB and picks up the scenarios and executes them. The variables and their defined values are picked up by the test framework and the test cases are executed for those values.

The test framework is considered the core of automation design. It subjects the test cases to different scenarios. The framework monitors the results of every iteration and the results are stored. The test framework contains the main logic for interaction, initiating and controlling all modules.

Tools and Results Modules:

When a test framework performs its operations, there are a set of tools that may be required. In order to run the compiled code, certain runtime tools and utilities may be required. For example, IP Packet Simulators or User Login Simulators or Machine Simulators may be needed. In this case, the test framework invokes all these different tools and utilities.

Report Generator and Reports/ Metrics Modules:

Preparing reports is a complex and time consuming effort and hence it should be part of the automation design. There should be customized reports such as an executive report, which gives very high level status; technical reports, which give a moderate level of detail of the test run. And detailed or debug reports which are generated for developers to debug the failed test cases and the product. The periodicity of the reports is different, such as daily, weekly, monthly, and milestone reports. Having reports of different levels of details and different periodicities can address the needs of multiple constituents and thus provide significant returns.

The module that takes the necessary inputs and prepares a formatted report is called a report generator. Once the results are available, the report generator can generate metrics.

3. a. List the requirements for testing tool.

Requirement 1: No hard coding in the test suite
Requirement 2: Test case/ suite expandability
Requirement 3: Reuse of code for different types of testing, test cases
Requirement 4: Automatic setup and cleanup
Requirement 5: Independent test cases
Requirement 6: Test cases dependency
Requirement 7: Insulating test cases during execution
Requirement 8: Coding standards and directory structure
Requirement 9: Selective execution of test cases
Requirement 10: Random execution of test cases
Requirement 11: Parallel execution of test cases
Requirement 12: Looping the test cases
Requirement 13: Grouping of test scenarios
Requirement 14: Test case execution based on previous results
Requirement 15: Remote execution of test cases
Requirement 16: Automatic archival of test data
Requirement 17: Reporting scheme
Requirement 18: Independent of languages
Requirement 19: Portability to different platforms

b. Discuss the challenges in automation.

1) Testing the complete application:

There are millions of test combinations. It's not possible to test each and every combination both in manual as well as in automation testing. If you try all these combinations you will never ship the product

2) Misunderstanding of company processes:

Some times you just don't pay proper attention what the company-defined processes are and these are for what purposes. There are some myths in testers that they should only go with company processes even these processes are not applicable for their current testing scenario. This results in incomplete and inappropriate application testing.

3) Relationship with developers:

Big challenge. Requires very skilled tester to handle this relation positively and even by completing the work in testers way. There are simply hundreds of excuses developers or testers can make when they are not agree with some points. For this tester also requires good communication, troubleshooting and analyzing skill.

4) Regression testing:

When project goes on expanding the regression testing work simply becomes uncontrolled. Pressure to handle the current functionality changes, previous working functionality checks and bug tracking.

5) Lack of skilled testers:

I will call this as 'wrong management decision' while selecting or training testers for their project task in hand. These unskilled fellows may add more chaos than simplifying the testing work. This results into incomplete, insufficient and ad-hoc testing throughout the testing life cycle.

6) Testing always under time constraint:

Hey tester, we want to ship this product by this weekend, are you ready for completion? When this order comes from boss, tester simply focuses on task completion and not on the test coverage and quality of work. There is huge list of tasks that you need to complete within specified time. This includes writing, executing, automating and reviewing the test cases.

7) Which tests to execute first?

If you are facing the challenge stated in point no 6, then how will you take decision which test cases should be executed and with what priority? Which tests are important over others? This requires good experience to work under pressure.

8) Understanding the requirements:

Some times testers are responsible for communicating with customers for understanding the requirements. What if tester fails to understand the requirements? Will he be able to test the application properly? Definitely No! Testers require good listening and understanding capabilities.

9) Automation testing:

Many sub challenges – Should automate the testing work? Till what level automation should be done? Do you have sufficient and skilled resources for automation? Is time permissible for automating the test cases? Decision of automation or manual testing will need to address the pros and cons of each process.

10) Decision to stop the testing:

When to stop testing? Very difficult decision. Requires core judgment of testing processes and importance of each process. Also requires 'on the fly' decision ability.

11) One test team under multiple projects:

Challenging to keep track of each task. Communication challenges. Many times results in failure of one or both the projects.

12) Reuse of Test scripts:

Application development methods are changing rapidly, making it difficult to manage the test tools and test scripts. Test script migration or reuse is very essential but difficult task.

13) Testers focusing on finding easy bugs:

If organization is rewarding testers based on number of bugs (very bad approach to judge testers performance) then some testers only concentrate on finding easy bugs those don't require deep understanding and testing. A hard or subtle bug remains unnoticed in such testing approach.

14) To cope with attrition:

Increasing salaries and benefits making many employees leave the company at very short career intervals. Managements are facing hard problems to cope with attrition rate. Challenges – New testers require project training from the beginning, complex projects are difficult to understand, delay in shipping date!

4. How to select the test tool and also discuss the criteria and steps in selecting the tool?

Selecting the test tool is an important aspect of test automation for several reasons as given below:

a. Free tools are not well supported and get phased out soon.
b. Developing in-house tools takes time.
c. Test tools sold by vendors are expensive
d. Test tools require strong training
e. Test tools generally do not meet all the requirements for automation
f. Not all test tools run on all platforms.

Criteria for selecting Test tools:

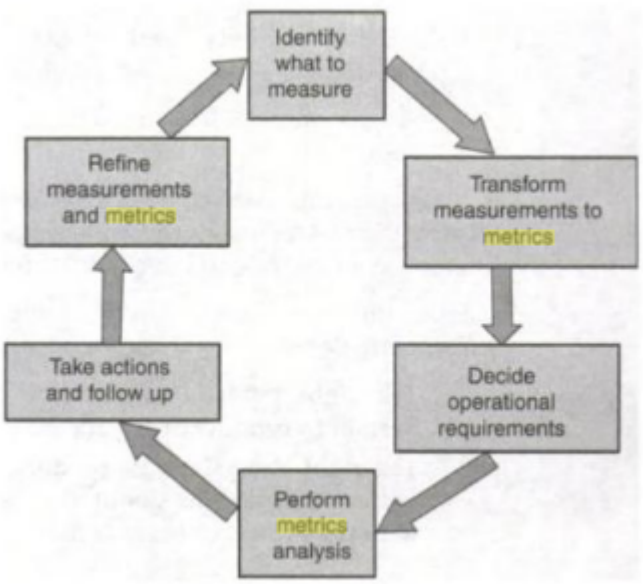
The criteria for selecting tools are classified into four categories. They are

a. Meeting requirements
b. Technology expectations
c. Training/skills; and
d. Management aspects

Steps for tool selection and deployment

- Identify your test suite requirements among the generic requirements discussed. Add other requirements
- Make sure experiences discussed are taken care of
- Collect the experiences of other organizations which used similar test tools
- Keep a checklist of questions to be asked to the vendors on cost/effort/support
- Identify list of tools that meet the above requirements
- Evaluate and shortlist one/set of tools and train all test developers on the tool
- Deploy the tool across test teams after training all potential users of the tool.

5. Explain in detail about the steps in metrics program.



The first step involved is to decide what measurements are important and collect data accordingly. The effort spent on testing, number of defects, and number of test cases, are some examples of measurements. Depending on what the data is used for, the granularity of measurement will vary.

The second step involved in metrics collection is defining how to combine data points or measurements to provide meaningful metrics.

The third step in the metrics program is deciding the operational requirements for the measurements. The operational requirement for a metrics plan should lay down not only the periodicity but also other operational issues such as who should collect measurements, who should receive the analysis, and so on.

The fourth step involved in a metrics program is to analyze the metrics to identify both positive areas and improvement areas on product quality

The final step involved in a metrics plan is to take necessary action and follow up on the action. The purpose of the metrics program is defeated if the action items are not followed through to completion.

Any metrics program is a continuous and ongoing process. As we make measurement s, transform the measurements into metrics, analyze the metrics, and take corrective action, the issues for which the measurements were made in the first place will become resolved. The metrics program continually go through the steps described above with different measurements or metrics.

6. Discuss on different project metrics.

Effort and schedule are the two factors to be tracked for any phase or activity. The effort is tracked closely and met, then the schedule can be met. The schedule can also be met by adding more effort to the project.

The basic measurements that are very natural, simple to capture and form the inputs to the metrics are

a. The different activities and the initial baselined effort and schedule for each of the activities; this is input at the beginning of the project/phase
b. The actual effort and time taken for the various activities; this is entered as and when the activities take place
c. The revised estimate of effort and schedule; these are re-calculated at appropriate times in the project life.

Effort variance (EV) calculates variance of actual effort versus planned effort. The formula for effort variance is:

Effort variance = [(Actual effort - Planned Effort)/Planned effort] * 100

The effort variance may be greater than expected.
For example, we estimated 100 hours but actual work took 110 hours.

In this case, there would be a +10% effort variance. Some of the causes why this positive variance might have occurred are:
- Estimation parameters were wrong.
- Test scope was not understood in totality.
- Underestimated the test group's inefficient process.
- The estimate was changed without changing scope just to make the numbers match the preferred schedule.
- Added functionalities that were not in the customer requirements.

The effort variance may be less than expected.
For example, we estimated that the test project will take 100 hours to complete but it actually took 90 hours.

This results in a -10% effort variance. Some of the causes why this negative variance might have occurred are:
- Estimation parameters were wrong.
- There was an improvement in the process
- Team performed much better than expected
- Testing was not finished and one or more of the requirements was missed.

Schedule Variance

Schedule Variance= Earned Value - Planned Value

Earned Value-> Budget value of the work completed in specified time

Planned Value -> Budget value of the work that was planned to be completed in specific time

The difference reflects the amount of budget that could not be completed or that was excessively done in specific time.

Effort Distribution across phase

From an estimate of the number of person-days required for the programming effort, and an estimate of the relative distribution of the effort by phase, it is possible to estimate the number of person-days of effort required for the total software development project.

From past project experience, the standard distribution of effort for large-scale software development is approximately:

Analysis	10 %	
Design	15 %	
Programming/unit testing	30 %	
System test	10 %	
Acceptance test	5 %	
Manual procedures	5 %	*
User training	5 %	*
Conversion	5 %	*
Technical support	5 %	*
Project management	10 %	**

As an example, if the estimate assumes 75,000 lines of code and a productivity rate of 30 lines/day from analysis through to implementation, the total effort would be calculated as 2,500 person-days (75,000/30). The corresponding effort by phase would be:

Analysis	10 %	250 days
Design	15 %	375 days
Programming/unit testing	30 %	750 days
System test	10 %	250 days
Acceptance test	5 %	125 days
Manual procedures	5 %	125 days
User training	5 %	125 days
Conversion	5 %	125 days
Technical support	5 %	125 days
Project management	10 %	250 days

Extending the estimated days by the projected daily rate would provide a ball-park cost for each phase.

Typically unbounded tasks with less correlation to lines of code. Refers only to the time of the Project Manager and is typically determined by the overall elapsed time of the project. Project leaders' time is included in the time for program/unit testing.

7. What are the different progress metrics? Brief.

Progress Metrics

Defects get detected by the testing team and get fixed by the development team.

Defect metrics are classified into test

- Defect metrics: Which helps the testing team in analysis of product quality and testing
- Development defect metrics: Which helps the development team in analysis of development activities.

A set of metrics that tracks how the different activities of the project are progressing

Test progress metrics capture the progress of defects found with time.

Test defect metrics

- A set of metrics helps to understand how the defects that are found can be used to improve testing and product quality.
- The defects are classified by defect priority and defect severity.
- Defect priority provides management perspective for the order of defect fixes.
- The severity of defects provides the test team a perspective of the impact of that defect in product functionality.

Defect priority and defect severity

Priority	**What it means**
1	Fix the defect on highest priority; fix it before the next built
2	Fix the defect on high priority before next test cycle
3	Fix the defect on moderate priority when time permits before the release
4	Postpone this defect for next release or live with this defects

Severity	**What it means**
1	The basic product functionality failing or product crashes
2	Unexpected error condition
3	A minor functionality is failing or behaves differently than expected
4	Cosmetic issue and no impact on the users

Defect Classification

Defect classification	**What it means**
Extreme	Product crashes or unusable. Need to be fixed immediately.

Critical	Basic functionality of the product not working. Need to be fixed before next cycle starts.
Important	Extended functionality of the product not working. Does not affect the progress of testing. Fix it before release
Minor	Product behaves differently. No impact on the team. Fix it when time permits.
Cosmetics	Minor irritant. Need not be fixed for this release

- Defect find rate
- Defect fix rate
- Outstanding defects rate
- Priority outstanding rate
- Defect trend
- Defect classification trend
- Weighted defects trend
- Defect cause distribution

Development Defect Metrics

While the defect metrics focuses on the number of defects, development defect metrics try to map those defects to different components of the product and to some of the parameters of development such as line of code.

- Component-wise defect distribution
- Defect density
- Defect removal rate
- Age analysis of outstanding defects
- Introduced and reopened defects trend

8. Explain the various productivity metrics.
Productivity Metrics

Productivity metrics combine several measurements and parameters with effort spent on the product. They help in finding out the capability of the team as well as for other purposes, such as,

- Estimating for the new release

- Finding out how well the team is progressing, understandingthe reason for variation in results.
- Estimating the number of defects that can be found
- Estimating the release date and quality
- Estimating the cost involved in the release.

Defect per 100 hours of testing

More testing reveals more defects. But there may be a point of diminishing returns when further testing may not reveal any defects.

If incoming defects in the product are reducing, it may mean various things.

- Testing is not effective
- The quality of the product is improving
- Effort spent in testing is falling

The metric defect per 100 hours of testing covers the third point and normalizes the number of defects found in the product with respect to the effort spent.

Defect per 100 hours of testing = (Total defect found in the product for a period/Total hours spend to get those defects)*100

Test cases executed per 100 hours of testing

The number of test cases executed by the test team for a particular duration depends on team productivity and quality of the product.

If the quality of the product is good, more test cases can be executed, as there may not be defects blocking the tests.

Test cases executed per 100 hours of testing helps in tracking productivity and also in judging the product quality.

Tets cases executed per 100 hours of testing = (Total test cases executed for a period/Total hours spent in test execution) * 100

Teat cases developed per 100 hours of testing

In a product scenario, not all the test cases qre written afresh for every release. New test cases are added to address new functionality and for testing features that were not tested earlier.

Existing test cases are modified to reflect changes in the product. Some test cases are deleted if they are no longer useful or if corresponding features are removed from the product.

The formula for test cases developed uses the count corresponding to added/modified and deleted test cases.

Test cases developed per 100 hours of testing = (Total test cases developed for a period/Total hours spent in test case development) *100

Defects per 100 test cases

The goal of testing is find out as many defects as possible. This is a function of two parameters,

- The effectiveness of the tests in uncovering defects
- The effectiveness of choosing tests that are capable of uncovering defects

The ability of a test cases to uncover defects depends on how well the test cases are designed and developed. But, in a typical product scenario, not all test cases are executed for every test cycle. Hence it is better to select that produce defects. A measure that quantifies these two parameters is defect per 100 test cases.

Defects per 100 test cases = (Toatl defects found for a period/Total test cases executed for the same period) *100

Defects per 100 failed test cases

Defect per 100 failed test cases is a good measure to find out how granular the test cases are. It indicates

- How many test cases neede to be executed when a defect is fixed.
- What defects need to be fixed so that an acceptable number of test cases reach the pass rate
- How the fail rate of test cases and defects affect each other for release readiness analysis.

Defects per 100 failed test cases = (Test defects found for a period/Total test cases failed due to those defects) * 100

Test phase effectiveness

- As testing is performed by various teams with objective of finding defects early at various phases, a metric is needed to compare the defects filled by each phase in testing.

- The defects found in various phases are, unit testing (UT), component testing (CT), integration testing (IT) and system testing (ST).

Closed defect Distribution

- The objective of testing is not only to find defects.

- The testing team also has the objective to ensure that all defects fond through testing are fixed so that the customer gets the benefit of testing and the product quality improves.

- To ensure the most of the defects are fixed, the testing team has to track the defects and analyze how they are closed. The closed defect distribution helps in this analysis.