NEW SYLLABUS T.Y. B.B.A. (C.A.)
CBCS PATTERN SEMESTER - VI

Software Testing

Dr. Aruna A. Deoskar Dr. Jyoti J. Malhotra

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

Syllabus ...

1. **Introduction** **[Lectures 10]**
   - 1.1 Introduction, Nature of Errors
   - 1.2 Testing Objectives
   - 1.3 Testing Principles
   - 1.4 Testing Fundamentals
   - 1.5 Software reviews, Formal Technical reviews
   - 1.6 Inspection and Walkthrough
   - 1.7 Testing Life Cycle
2. **Approaches to Testing - Testing Methods** **[Lectures 5]**
   - 2.1 White Box Testing and types of White Box Testing
   - 2.2 Test Case Design
   - 2.3 Black Box Testing and types of Black Box Testing
   - 2.4 Gray Box Testing
3. **Software Testing Strategies and Software Metrics** **[Lectures 10]**
   - 3.1 Software Testing Process
   - 3.2 Unit Testing
   - 3.3 Integration- Top-down, Bottom up
   - 3.4 System Testing
   - 3.5 Acceptance Testing (Alpha, Beta Testing)
   - 3.6 Validation and Verification
   - 3.7 Big Bang Approach
   - 3.8 Sandwich approach
   - 3.9 Performance Testing
   - 3.10 Regression Testing
   - 3.11 Smoke Testing
   - 3.12 Load Testing
4. **Software Metrics** **[Lectures 10]**
   - 4.1 Introduction
   - 4.2 Basic Metrics - Size-oriented metric, Function - oriented metric
   - 4.3 Cyclometic Complexity Metrics
     Examples on Cylcometic Complexity
5. **Testing for Specialized Environments** **[Lectures 5]**
   - 5.1 Testing GUI's
   - 5.2 Testing of Client/Serve Architectures
   - 5.3 Testing Documentation and Help Facilities
   - 5.4 Testing for Real-Time Systems
6. **Testing Tools and Software Quality Assurance (Introduction)** **[Lectures 8]**
   - 6.1 JUnit, Apahe JMeter, Win runner
   - 6.2 Load runner, Rational Robot
   - 6.3 Quality Concepts, Quality Movement, Background Issues, SQA activities
   - 6.4 Formal approaches to SQA
   - 6.5 Statistical Quality Assurance
   - 6.6 Software Reliability
   - 6.7 The ISO 9000 Quality Standards
   - 6.8 SQA Plan
   - 6.9 Six sigma
   - 6.10 Informal Reviews

Contents ...

1... Introduction

Learning Objectives ...

- To study the concept of the Software Testing.
- To know about the nature of errors.
- To learn about the Testing Principles and Testing fundamentals.
- To get information about the debugging.

1.1 INTRODUCTION TO SOFTWARE TESTING

Definition: [S-22, 23; W-22]

- "Software testing is a process used to identify the correctness, completeness and quality of developed software".
- Testing is the process of exercising and evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements.
- Primary role of testing is not demonstration of correct performance, but the exposure of hidden defects. - Glen Myers
- Testing is concerned with *errors, faults, failures and incidents*. A test is the act of exercising software with an objective of :
  - Finding failure
  - Demonstrate correct execution - Paul Jorgensen
- According to ANSI/IEEE 1059 standard, Testing can be defined as - A process of analyzing a software item to detect the differences between existing and required conditions (that is defects/errors/bugs) and to evaluate the features of the software item.
- Software testing is a process used to identify the correctness, completeness, and quality of developed computer software. It includes a set of activities conducted with the intent of finding errors in software so that it could be corrected before the product is released to the end users.

OR

- In simple words, software testing is an activity to check whether the actual results match the expected results and to ensure that the software system is defect free.

OR

- It can also be stated as the process of validating and verifying that a software program or application or product:
  - Meets the business and technical requirements that guided it's design and development.
  - Works as expected.
  - Can be implemented with the desired characteristics.

Benefits of Software Testing:

1. It improves the Quality as testing looks for a class of problems and flags them as possible candidates for investigation and fixes.
2. Testing demonstrates that software functions appear to be working according to specifications.

Goals of Software Testing:

- The main goal of software testing is to find bugs as early as possible and to fix bugs and to make sure that the software is bug free.
- The goals of software testing may be classified into three major categories as shown in Fig. 1.1.

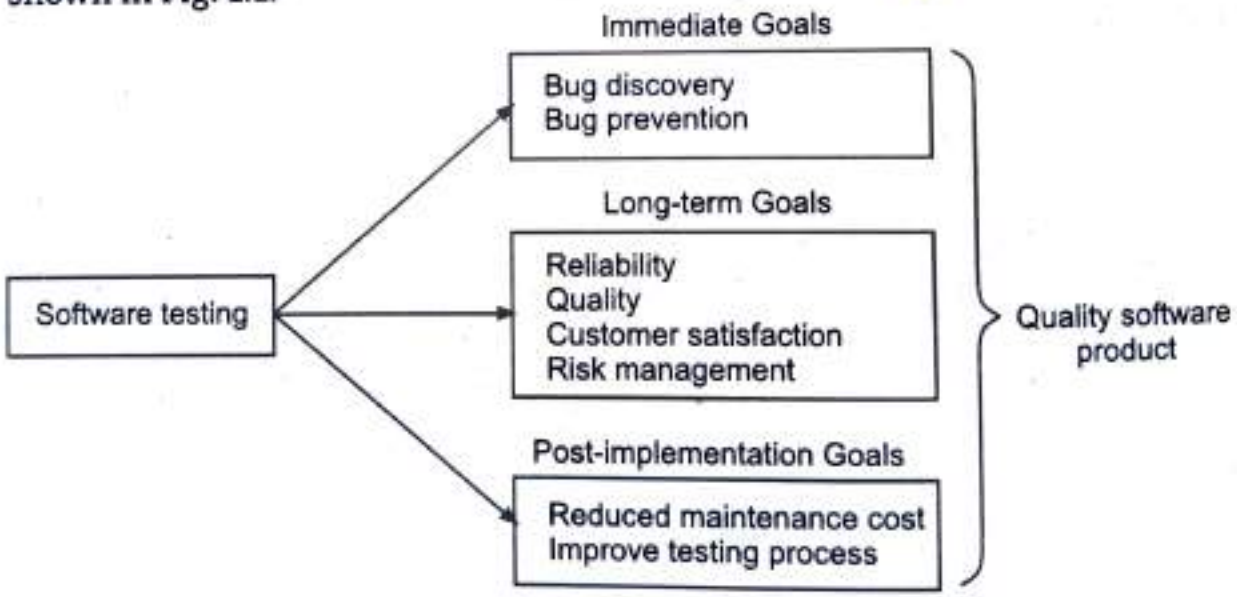


Fig. 1.1: Software Testing Goals

1. **Immediate Goals:** These goals are also called as short term goals. These testing goals are the immediate result after testing. These goals contain:
   (i) **Bug discovery:** This is the immediate goal of software testing to find errors at any stage of software development. Numbers of the bugs are discovered in the early stage of testing.
   (ii) **Bug prevention:** This is the resultant action of bug discovery.

2. **Long Term Goals:** These testing goals affect the software product quality in the long term. These include:
   (i) **Quality:** This goal enhances the quality of the software products.
   (ii) **Customer satisfaction:** This goal verifies the customer's satisfaction for developed software product.
   (iii) **Reliability:** It is a matter of confidence that the software will not fail. In short reliability means to gain the confidence of the customers by providing them a quality product.
   (iv) **Risk management:** Risk must be done to reduce the failure of product and to manage risk in different situations.
3. **Post Implemented Goals:** These goals are important after the software product released. Some of them are listed below:
   (i) **Reduce maintenance cost:** Post-released errors are costlier to fix and difficult to identify.
   (ii) **Improved software testing process:** These goals improve the testing process for future use of software projects. These goals are known as post-implementation goals.

1.1.1 Nature of Errors [S-22, 23; W-22]

Software Bug:

- A software bug is the common term used to describe an error, flaw, mistake, failure or fault in a computer program or system that produces an incorrect or unexpected result, or which causes the program to perform in an un-intended manner.
- Most bugs arise from mistakes and errors in either a program's source code or its design, but the main cause can be traced to the specification.
- A Software bug occurs when one or more of the following is true:
  - The Software does not do something that the product specification says it should do.
  - The Software does something that the product specification says it should not do.

1. **Errors:**

- The system is in a state such that further processing by the system will lead to a failure.
- An error is a **mistake**, **misconception** or **misunderstanding** on the part of a software developer. It might be typographical error, a misleading of specifications, or a misunderstanding of what a subroutine does.
- The errors may be:
  - Actual bugs in the code.
  - Incorrect implementation of the requirements or functional specifications because of Misunderstandings and/or Incomplete requirements or functional specifications.

- Incorrect human action that produces erroneous step, process, or inaccurate result.
- User interface errors i.e. Functionality, Communication, Command structure, Missing commands, Performance, Output.
- Calculation errors.
- Errors in handling or interpreting data.
- Documentation - The user does not observe the ++operation described in manuals.
- Testing errors.

2. Faults:

- A fault occurs when a human error results in a mistake in some software product(s).
- For example, a developer might misunderstand a user-interface requirement and therefore create a design that includes misunderstanding. The design fault can also result in incorrect code, as well as incorrect instructions in user manual.

Fig. 1.2: Flow of Fault

- A fault is the difference between an incorrect program and the correct version. A single error can result in one or more faults. One fault can result in multiple changes to one product (such as changing several sections of a piece of code) or multiple changes to multiple products (such as change to requirements, design code and test plans etc).

Examples of Faults and Errors:

Faults in the Interface specification:

- Mismatch between what the client needs and what the server offers.
- Mismatch between requirements and implementation.

Algorithmic Faults:

- Missing initialization.
- Branching errors (too soon, too late).

Mechanical Faults (very hard to find):

- Documentation does not match actual conditions or operating procedures.

Errors:

- Stress or overload errors.
- Capacity or boundary errors.
- Timing errors.
- Throughput or performance errors.

3. Failure:

- A fault (bug) can go undetected until a failure occurs, which is when a user or tester perceives that the system is not delivering the expected service.
- Failures can be defined as :
  - Deviation of the system from its expected behavior or service.
  - The inability of a system or component to perform its required functions within specified performance requirements.
- Failures can be discovered both before and after system delivery, as they can occur in testing as well as in operation.
- Faults represent problems that the developer sees, while failures are problems that the user sees.
- Ideally the life of a bug ends when it is uncovered in testing and fixed.

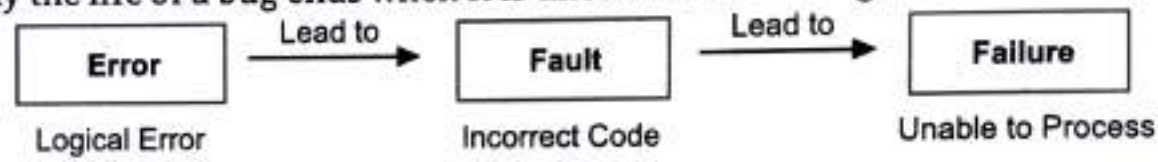


Fig. 1.3: Flow of Failure

- For example, consider ATM machine. Developer might have forgotten to fire update query on Database when user changes the PIN number, and screen displays the message "Your PIN number has been changed!!" When user tries to access the system with new PIN number, System gives the message "Sorry unable to process".

Defects:

- Defects are :
  - Abnormal behavior of the software.
  - Non-conformance to requirements or functional/program specification.
  - Errors in Testing.
  - Mistakes in Correction.
- For example, Requirements and specification defects, design defects, coding defects or testing defects.

1.2 TESTING OBJECTIVES

- Testing is a process of executing a program with the intention of finding Errors.
- Establishing confidence that a program does what it is supposed to do. It is the measurement of Software Quality which confirms that a program performs its intended functions correctly.
- Testing identifies the difference between Expected and Actual Result. This is a process of trying to discover every conceivable fault or weakness in a work product.

Basic objectives of Testing are:

- To find a bug/ any defect/errors in Software.

- To ensure the Software quality which confirms that a program performs its intended functions correctly?
- To explore the weaknesses/limitations of a software.
- To ensure the Software quality in terms of its reliability and working as per the customer satisfaction and gives expected results.
- To verify the software as per the standards.
- To validate the software as per the customer requirements.

- It is good to begin testing from the first stage, to avoid complexity by fixing the errors at the last stage.

We do software testing for many reasons, such as:

1. To check the reliability of the software.
2. To be ensured that the software does not contain any bug which can become a reason for failure?
3. To check the software was made according to its specification.
4. To check that the software meets its requirements.
5. To check that users are capable of using the software.
6. To check software works with other software and hardware it needs to work with.
7. To improve the quality of software by removing maximum possible errors or defects from it.

1.3 TESTING PRINCIPLES [S-22, 23; W-22]

- As software development techniques have advanced during the last decades, some basic principles of testing have also been established. Describing theoretical ideas and practical hints, these principles can be seen as a basic guideline for both testing and coding.

1. Testing shows the presence of errors:

- Testing an application can only disclose one or more defects that exist in the application, however, testing alone cannot prove that the application is error free. It is important to design effective test cases which find as many defects as possible.

2. Exhaustive testing is impossible:

- Unless the application under test (UAT) has a very simple logical structure and limited input, it is not possible to test all possible combinations of data and scenarios. For this reason, risk and priorities are used to concentrate on the most important aspects to test.

3. Early testing:

- We can start testing as soon as the initial products, such as the requirement or design documents are available. It is common for the testing phase to get squeezed at the end of the development lifecycle, i.e. when development has finished, so by starting testing early, we can prepare testing for each level of the development lifecycle.

- When defects are found earlier in the life cycle, they are much easier and cheaper to fix. It is much cheaper to change an incorrect requirement than having to change functionality in a large system that is not working as requested or as designed!

4. **Defect clustering:**

- During testing, it can be observed that most of the reported defects are related to a small number of modules within a system i.e. a small number of modules contain most of the defects in the system.

5. **The Fading effectiveness:**

- If you keep running the same set of tests over and over again, there chances are that no more new defects will be discovered by those test cases. Because as the system evolves, many of the previously reported defects will have been fixed and the old test cases do not apply anymore. Anytime a fault is fixed or a new functionality added, we need to do regression testing to make sure the new changed software has not affected any other part of the software.

6. **Testing depends on context:**

- Different methodologies, techniques and types of testing are related to the type and nature of the application. For example, a software application in a medical device needs more testing than games software. More importantly a medical device software requires risk based testing, be compliant with medical industry regulators and possibly specific test design techniques. By the same token, a very popular website needs to go through rigorous performance testing as well as functionality testing to make sure the performance is not affected by the load on the servers.

7. **Absence of errors myth:**

- Just because testing did not find any defects in the software, it does not mean that the software is ready to be shipped. Were the executed tests really designed to catch the most defects? Or where they designed to see if the software matched the user's requirements? There are many other factors to be considered before making a decision to dispatch the software.

1.4 TESTING FUNDAMENTALS

- In testing, we can describe goals as intended outputs of the software testing process. Primary goals of software testing are:
  1. Error detection.
  2. Debugging.
  3. Verification and Validation.
  4. Test coverage.
  5. Quality and Reliability.

1. **Error detection:**

- An error is a human action producing an incorrect result. When programmers make errors, they bring in faults to program code. For Example,
  - Incorrect usage of software by users.
  - Bad architecture and design by architects and designers.
  - Bad programming by developers.
  - Inadequate testing by testers.
  - Wrong build using incorrect configuration items by Build Team Member.
- Software error detection is one of the most challenging problems in software engineering. It involves identifying bugs/errors/defects in software without correcting it. Normally professionals with a quality assurance background are involved in bug's identification. Test execution is performed in the testing phase.

2. **Debugging:**

- Debugging occurs as a consequence of successful testing. When testing detects an error, debugging is an action that results in finding the cause of failure and removing it. It determines why a system doesn't meet one or more of its specifications or requirements and making it meet those specifications. Debugging is not testing but testing can be a part of debugging. We will see more on debugging in the next section.

3. **Verification and Validation:**

Verification:

- Verification is the process of confirming whether software meets its specifications. This process of reviews deliverables throughout the life cycle. It examines Product Requirements, Specifications, Design, code for errors, faults and failures. It is usually performed by STATIC testing, or inspecting without execution on a computer. Inspections, walkthroughs and reviews are examples of verification techniques.
- Verification is the assurance that the products of a particular phase in the development process are consistent with the requirements of that phase. It is a low level activity. In simple terms, verification makes sure that the "product is being built right".
- For example, one of the purposes of system testing is to give assurance that the system design is consistent with the requirements of the system design phase.

Validation:

- Validation is a process of confirming whether software meets a user's requirements. This process covers executing something to see how it behaves. Unit, Integration, System, and Acceptance testing are examples of validation techniques. Validation is usually performed by DYNAMIC testing, or testing with execution on a computer.
- Black box testing and White box testing are examples of test case design techniques. Validation is the assurance that the final product satisfies the system requirements.

It is high level activity. In simple terms, validation makes sure that the "right product is being built".

- The purpose of validation is to ensure that the system has implemented all requirements, so that each function can be traced back to a particular customer requirement. Table 1.1 summarizes the difference between Verification and Validation.

Table 1.1: Difference between Verification and Validation [W-22; S-23]

Sr. No.	Verification	Validation
1.	It is a process of confirming whether software meets its specifications.	It is a process of confirming whether software meets the user's requirements.
2.	It is the process of reviewing/ inspecting deliverables throughout the life cycle.	It is the process of executing something to see how it behaves.
3.	It checks product requirements, specification, design, code for errors, faults and failures.	It examines the entire product for errors, fault and failures against the verified requirements, specifications and design.
4.	Verification is usually performed by Static testing, or inspecting without execution on a computer.	Validation is usually performed by Dynamic testing, or testing with execution on a computer.
5.	*Verification* is the assurance that the products of a particular phase in the development process are consistent with the defined standard.	*Validation* is the assurance that the final product satisfies the system requirements.
6.	It is a low level activity.	It is a high level activity.
7.	Verification comes into picture during initial phases of SDLC.	Validation comes into picture during later stages of SDLC (once code is ready).
8.	Verification methods : ▪ Inspections ▪ Walkthroughs ▪ Reviews	Validation methods : ▪ White box testing ▪ Black box testing
9.	Verification makes sure that the "product is being built right".	Validation makes sure that the "right product is being built".
10.	Static mode of testing.	Dynamic mode of testing.

- Debugging, testing, and verification are mapped to the software life cycle as shown in Fig. 1.4.

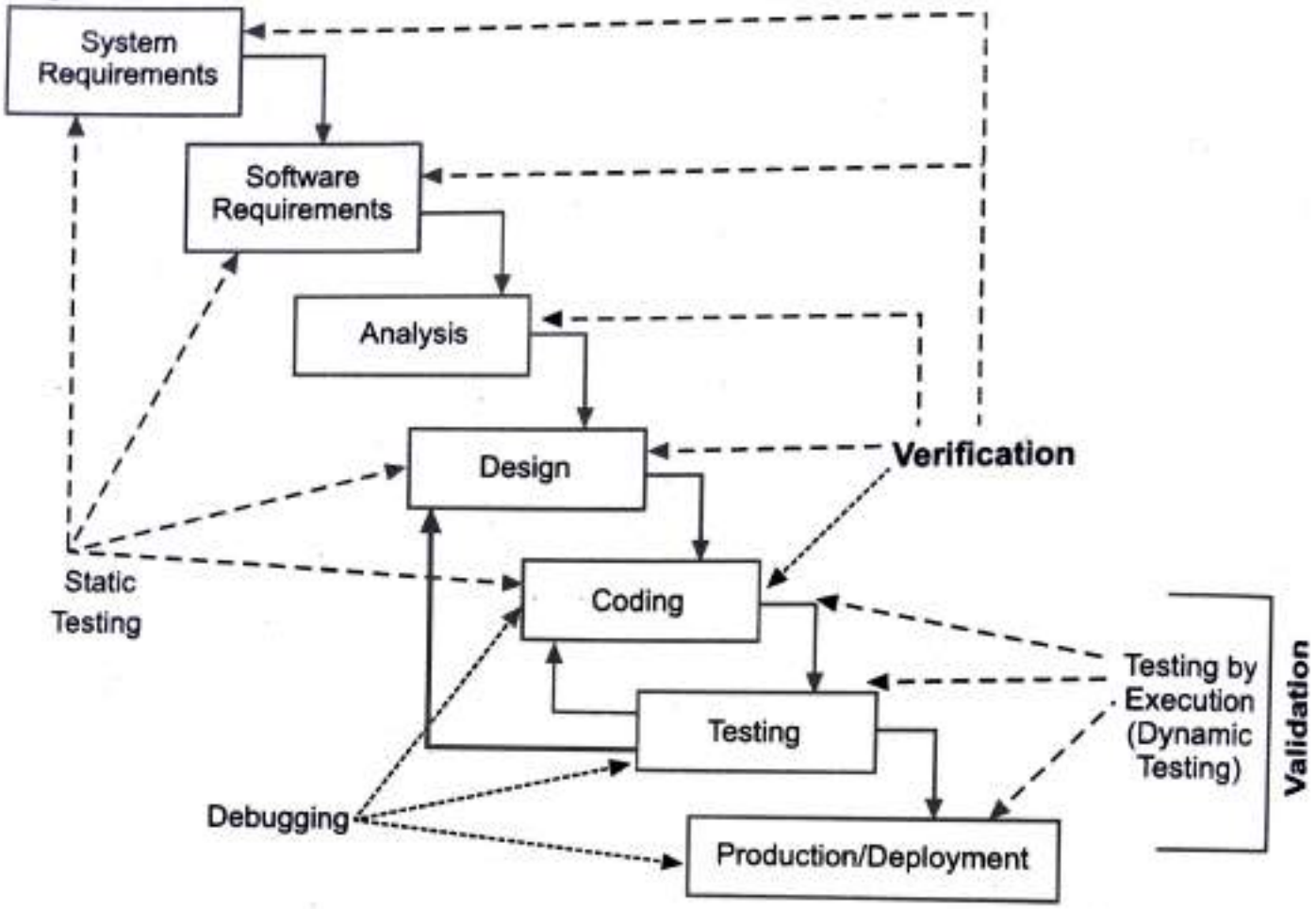


Fig. 1.4: Debugging, Verification and Testing

4. Test Coverage:

- Exhaustive testing is impossible, not everything can be tested. Not even a significant subset of everything can be tested. Therefore, testing needs to assign tasks reasonably and prioritize thoroughly. Generally, every feature should be tested at least with one valid input case.
- We can also test input permutations, invalid input, and non-functional requirements depending upon the operational profile of software. Highly present and frequent use scenarios should have more coverage than infrequently encountered and insignificant scenarios.

5. Quality and Reliability:

- Software quality is the degree of conformance to explicit or implicit requirements and expectations.
  - *Explicit*: Clearly defined and documented.
  - *Implicit*: Not clearly defined and documented but indirectly suggested.
  - *Requirements*: Business/product/software requirements.
  - *Expectations*: End-user expectations.

- Reliability is one aspect of quality. To ensure that a program is of high quality and is reliable, a software tester must do both verify and validate throughout the product development process.

1.4.1 Need of Software Testing

- The main benefit of testing is the identification of errors and later removal of the errors.
- We assume that our work may have errors/mistakes; hence we all need to check our own work. However, some mistakes come from bad assumptions and blind spots. Ideally, we should get someone else to check our work because another person is more likely to spot the flaws.

Example:

- Assume you are using a net banking application to transfer the amount to your friend's account. So, you start the transaction, get a successful transaction message, and the amount also deducts from your account. However, your friend confirms that his/her account has not received any credits yet. Similarly, your account is also not reflecting the reversed transaction. This will certainly make you upset and leave you as an unsatisfied customer.
- This would be raising the following issues:
  - Security or user credential issues.
  - Authentication issues.
- This is one situation where testing the software is very important before installing it. Testing cannot be ignored because it impacts all the end users of software.
- If the website would have got tested thoroughly for all the possible user operations this problem would have been found weli in advance and got fixed before organizing the website.

Reasons to use Software testing:

- Software testing is very important because of the following reasons :
  1. Software testing is required to find out the defects and errors that were made during the development phases.
  2. It is essential since it makes sure of the Customer's reliability and their satisfaction in the application.
  3. It is very important to ensure the Quality of the product. Quality product delivered to the customers helps in gaining their confidence.
  4. Testing is necessary in order to provide the facilities to the customers like the delivery of high quality product or software application which requires lower maintenance cost and hence results into more accurate, consistent and reliable results.

1.4.2 Model for Software Testing

- Testing should be performed in a planned way. For the planned execution of a testing process, we need to consider every element and every aspect related to software testing.
- For this reason, we consider the related elements and team members involved in the testing model, as shown in Fig. 1.5.

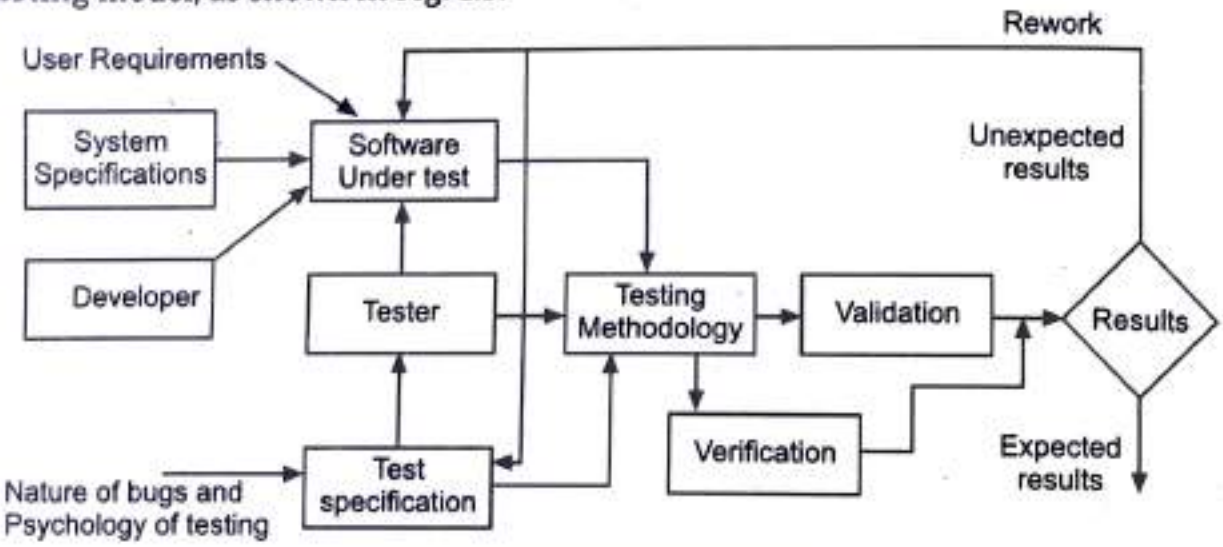


Fig. 1.5: Software Testing Model

- The software is basically a part of a system for which it is being developed or produced. Systems consist of hardware and software to make the software product run. The software developer develops the software in the prescribed system environment considering the testability of the software. Testability is a major issue for the developer while developing the software. As badly written software may be difficult and critical to test.
- Software testers are supposed to get on with their tasks as soon as the requirements are specified. Testers work on the basis of a test specification which specifies bugs based on the criticality. Based on the software type and the specifications, software testers decide a testing methodology which could be verification and/or validation testing and it guides how the testing is performed.
- With suitable or appropriate testing techniques decided in the testing methodology, testing is performed on the software with a particular goal objective. If the testing results are in line with the desired goals, then the testing is successful; otherwise, rework has to be done on software until the desired results are achieved.

1.4.3 Types of Testing [S-22]

- There are different kinds of testing that focus on finding different kinds of Software errors as shown in Fig. 1.6.

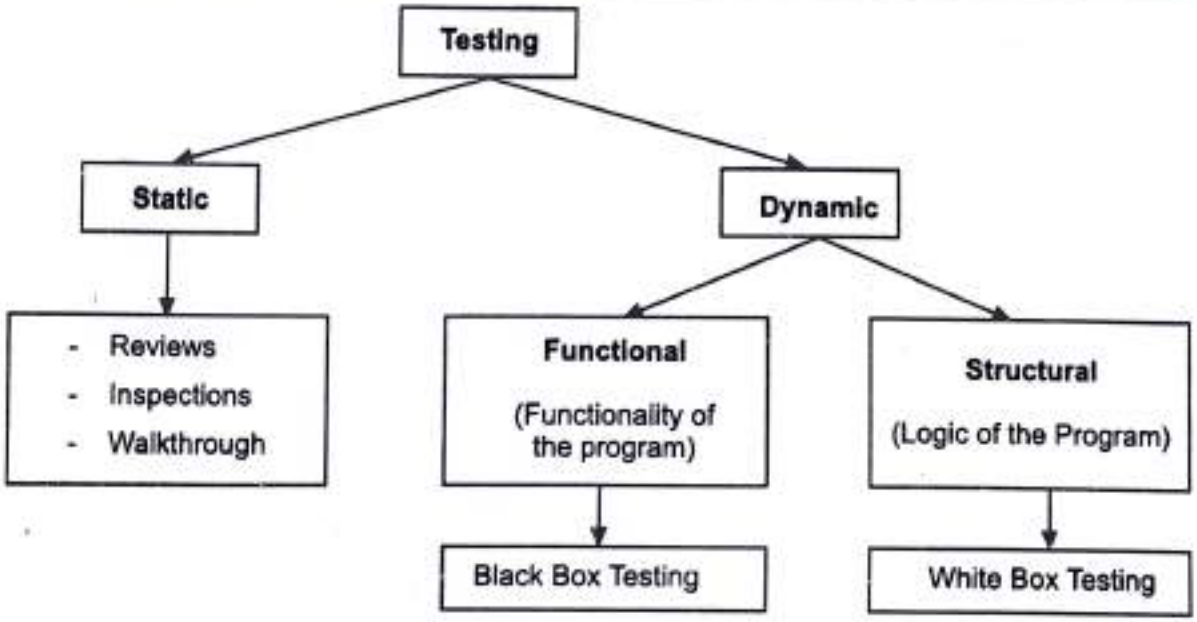


Fig. 1.6: Types of Testing

1. **Static Testing:**

- Static Testing also called as "**Dry Run Testing**" refers to test something, which is STATIC - not running. For example, consider an analogy that you want to purchase the vehicle. The Static tests can be:
  - Door Locks and Door Retention Components.
  - Seating Systems.
  - Seat Belt Assembly.
  - Kicking the tires.
  - Checking the paint.
- Static testing is a type of testing which requires only the source code of the product, not the binaries or executables. It does not involve executing the programs, but involves people going through the code to find out whether:
  - The code works according to the functional requirement.
  - The code has been written in accordance with the design developed.
  - The code for any functionality has been missed out.
  - The code handles errors properly.
- Static testing may not only examine the code, but also specification, design and user documents. It is testing prior to execution. Static testing is done by humans or with specialized tools. This includes desk checking of the code, code reviews, design walkthroughs, Inspections to check requirements and design documents and code standards.
- The Static testing technique can be done in two ways, Review and Static Analysis.
- Reviews, Walkthroughs, Inspections are the methods for detailed examination of a product by systematically reading the contents on a step-by-step basis to find defects.

Running syntax and type checkers as part of the compilation process, or other code analysis tools are part of static analyzer.

Syntax checking and manually reading the code to find errors is the method of static testing. This type of testing is mostly done by the developer himself. Static testing is usually the first type of testing done on any system.

2. **Dynamic Testing:**

Dynamic Testing refers to test the Software (code) through **executing it**. Testers generally do Dynamic Testing. All testing tools come under this category.

For example: Consider the same analogy of purchasing the vehicle. The dynamic testing can be,

- Starting it up.
- Listening to engine, horn.
- Driving down the road.

Dynamic Testing includes:

- Feed the program with input and observe behavior.
- Check a certain number of input and output values.

Dynamic Testing techniques:

- Unit Testing.
- Integrated Testing.
- System Testing.
- User Acceptance Testing.

- **Black-Box testing**: In Black-box testing, software is considered as a Box. Tester only knows what the software is supposed to do – he can't look in the box to see how it operates.
- **White Box Testing:** In White-box testing, tester has access to the program's code and can examine it for clues to help him with his testing. Types of white box testing are: statement coverage, branch coverage and path coverage.
- Black-box testing and White-box testing are covered in detail in Chapter 2

Functional Testing:

- Functional testing is the process of validating an application or Web site to verify that each feature complies with its specifications and correctly performs all its required functions. This involves a series of tests, which tests the functionality by using a wide range of normal and erroneous input data.
- This can involve testing of the product's user interface, APIs, database management, security, installation, networking; etc testing can be performed on an automated or manual basis using black-box methodologies.

1.5 SOFTWARE REVIEWS, FORMAL TECHNICAL REVIEWS

- Review is a kind of meeting for analyzing the product being developed or already developed. It is a verification process where the product is checked for defined specifications. Reviews can be done in various ways. It is a kind of static testing.
- As per IEEE definition – **"Review is a formal or informal meeting at which a product or document is presented to the user, customer, or other interested parties for comment and approval."**

1.5.1 Reviews

- **Review** is the name given to a testing technique done by people. It is a people based static technique in which one or more people are involved in examining certain activity or activities. There are a variety of ways in which reviews can be carried out across different organizations. Some reviews are done in a formal way, some are very informal and some reviews lies between these two techniques. Review technique can be performed individually or in groups.
- Review finds faults in specifications and other documents well in advance before they are used in subsequent phases. These faults detected through review can be fixed before their use in the next phase of development. Reviews are cost effective.
- Review in code is required for:
  - The necessary improvements in the product well in advance.
  - To bring simplicity and comfort during the development process, review helps in understanding the code complexity.
  - Review process reduces the cost impact of Software defects.
  - Review helps in defect amplification, defect detection and its removal.
  - Review process moves throughout the development phases. Review is done in terms of Requirement Review, Design review, Technical and code review.
- Software review method include following techniques:
  - Formal Review / Technical Review / Peer Review.
  - Informal Review.
  - Walkthrough.
  - Inspection.
  - Audit.

1.5.2 Formal Review Technique

- As its name implies, it is a formal review process to verify all related documents. Also known as technical review or peer review who has a high degree of formality while performing the review. This type of review mainly focuses on the technical issues and technical documents. A peer review would exclude managers from the review process. The success of this type of review heavily depends upon the review members

and their individual focus and knowledge. They can be very effective and useful if planned in a systematic manner, otherwise sometimes they are very wasteful especially if the meetings are not well disciplined. This level of documents needs to be planned effectively and so will have some set of documents. It is mandatory under the formal review process that even for a list of issues to be discussed or raised during a meeting should also be well documented.

- The formal review is very useful in finding important faults along with resolving difficult technical problems related to important technical decisions.
- The formal review process includes review planning, review execution and the post review follow up. Review process starts with listing the review participants which include review leader, 2 or 3 review team members having good knowledge of document under review and a scribe (could be a review team member) to note down the raised issues and corresponding decisions made by the reviewers, so that during post review follow up process they can be verified.
- Activities involved in FTR are:
  1. Review guidelines.
  2. Review Meeting.
  3. Review reporting and Record Keeping.
  4. Review checklist.

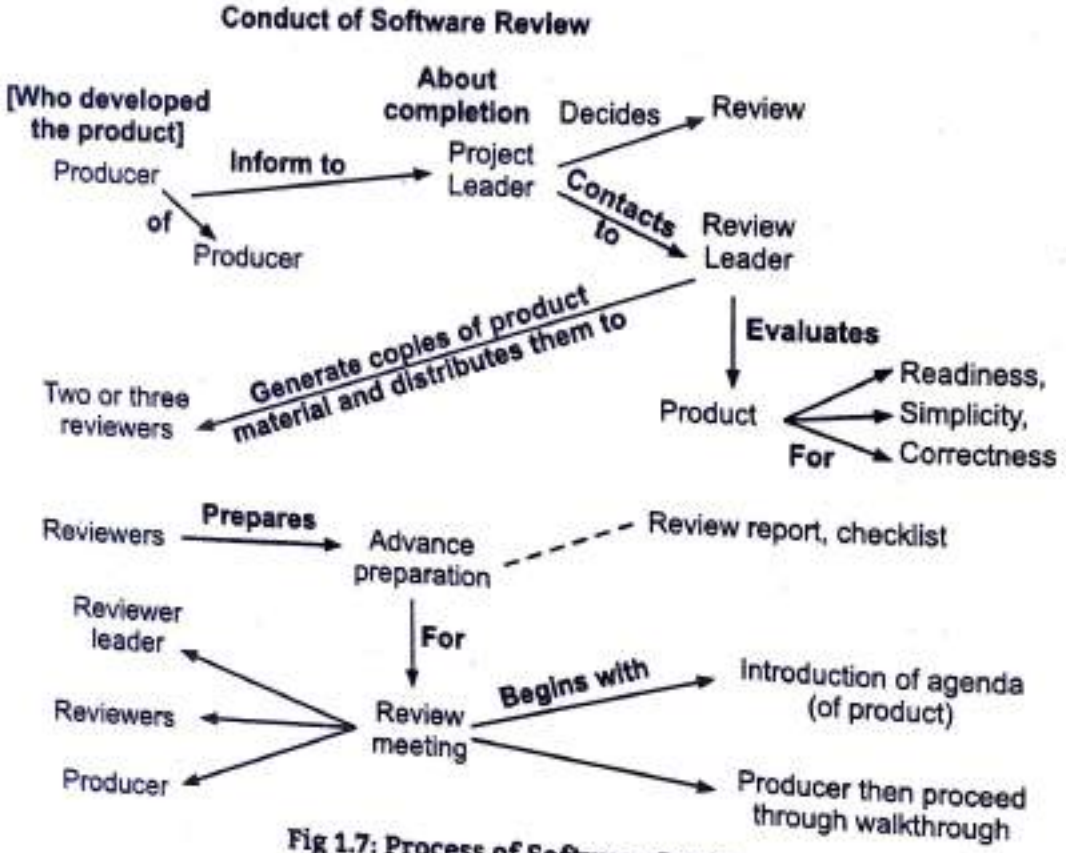


Fig 1.7: Process of Software Review

1.6 INSPECTION AND WALKTHROUGH

1.6.1 Inspection

- Inspection is a "formal evaluation technique in which software requirements, design, or code are examined in detail by a person or group other than the author to detect faults, violations of development standards, and other problems" (IBM).
- A software formal inspection is defect detection, defect elimination, and correction verification process carried out during the development life cycle. The goal of formal inspections is to ensure that defects are fixed early in the life cycle rather than late, when they are harder to find and more costly to fix. Formal inspections are held throughout the software development life cycle phases.

Activities:

- Inspection is a powerful review technique to support following activities:
  - Detect, identify, and describe defects.
  - Collect metrics data.
  - Verify "fitness for use" in subsequent efforts.
  - Provide controls that determine the next review step.

Requirements:

- Essential requirements for Inspection process include:
  - Definition of development process, entry criteria and exit criteria.
  - Description of the inspection process.
  - Correct execution of the process.
  - Authentic Documentation of Inspection process.
- Entry and exit criteria are used to determine if a product is ready to be inspected (entrance) and has successfully completed the process (exit). Entrance criteria to be satisfied depend on the product, but generally involve a determination that the product is mature enough to be used as is after defects are removed. Exit criteria are used mainly to assure that major defects found during the inspection process have been corrected. Correction of minor defects, those that will have no impact on the use of the product, may not be required by exit criteria. All work on the product to be inspected should cease once it has been submitted for inspection, otherwise the purpose of the inspection process will be defeated. Thus, the inspection must be scheduled in a timely manner.

Types of Inspections:

- Common types of inspections are:
  1. **High-level design inspection (I_0):** High Level Design (HLD) is the overall system design covering the system architecture and database design. It describes the relation between various modules and functions of the system. Data flow, flow charts and data structures, ERD are covered under HLD. Converted from SRS.

2. **Low-level design inspection (I_1):** Low Level Design (LLD) is like detailing the HLD. It defines the actual logic for each and every component/module of the system. Class diagrams with all the methods and relation between classes come under LLD. Program specificationss are covered under LLD.
3. **Code inspection (I_2):** Code Inspection (CI) is the inspection process of code developed by the developer based on design specifications. The inspection team reviews the code for its flow, complexity and development as per the need and specifications.

Participants of inspection team:

- Formal inspections are performed by a team (a moderator, reader, recorder, author, and other inspectors). The actual team size should consist of four to seven persons, although a minimum of three is acceptable. Each team member has a specific defined role. In addition, it is the responsibility of the entire inspection team to find and report defects; therefore, all members of the team are considered inspectors.

Difference between Formal inspections and other peer review processes:

- Formal inspections are distinguished from other types of peer reviews in that they follow a well-defined, tried, and proven process for evaluating, in detail, the actual product or partial product with the intent of finding defects. They are conducted by individuals from development, test, and assurance, and may include users.
- Formal inspections are more rigorous and well-defined than other peer review processes such as walkthroughs, and significantly more effective. Inspections do not take the place of milestone reviews, status reviews, or testing.

Goals:

- Inspections should be used to judge the quality of the software work product, not the quality of the people who create the product. For this reason, managers should neither participate in nor attend the inspection meeting.
- In addition, the results of inspections should be presented to management either statistically or as results for groups of products. This grouping of results will show managers the value of the inspection process without singling out any author.
- Using the inspection process to judge the capability of authors may result in less than honest and thorough results; that is, co-workers may be reluctant to identify defects if finding them will result in a poor performance review for a colleague.

1.6.2 Walkthrough

- Walkthroughs can be viewed as presentation reviews in which a review participant usually the developer of the software being reviewed, narrates a description of the software and the remainder of the review group provides their feedback throughout the presentation.
- These are referred to as presentation reviews because the bulk of the feedback usually only occurs for the material actually presented at the level it is presented.

- Thus, advance preparation on the part of reviewers is partially required during a walkthrough.
- Walkthroughs suffer from these limitations as well as the fact that they may lead to disorganized and uncontrolled reviews.
- Walkthrough is a Review process in which a participant gives the software description while the other members provide feedback throughout the presentation. Here, the development team members are involved in the review process.

Walkthrough Process:

- **Participants:**
  - Reviewer
  - Review team
- **Timing:** Completion of major milestones
- **Plan**
  - Identify walkthrough team
  - Schedule time, place
  - Distribute materials in advance
- **Execution**
- **Overview:** An overview presentation is made by the author of work product.
- **"Walkthrough" software**
- **Write report**
  - Identify walkthrough team
  - Identify item reviewed
  - State objectives
  - List deficiencies and recommendations
- **Implementation challenges**
  - Find, don't fix
  - Ego
  - Organization environment
- Walkthrough is an informal way of verification. The review team inspects the software in an informal way and in friendly environment. The team prepares the report after going through the software. The walkthrough team fixes up the schedule and time with the project team for walkthrough process with mutual consent. They distribute the project information in advance to all team members and then execute the walkthrough process. After the review a report is prepared. Report starts with the objective of walkthrough process and major observations. In report, the team mentions all the items which they have reviewed along with the observed deficiencies and recommendations. Since Walkthrough is executed in a friendly environment where the project team is also present, it suffers with many implementation challenges. The report might be prepared considering friendly gesture, ego or organization environment.

[S-22, 23; W-22]

1.7 TESTING LIFE CYCLE

- Once the project is confirmed to start, project development can be divided into following phases:
  - Software requirements phase.
  - Software Design.
  - Implementation.
  - Testing.
  - Maintenance.
- These phases form a model called as a waterfall model, a linear sequential model or software development life cycle, which suggests a systematic, sequential approach to software development.
- **Software Testing Life Cycle (STLC)** model is basically developed to identify which testing activities needs to be carried and what's the best time to perform them to accomplish those test activities. Even though testing differs between organizations, there is a testing life cycle.

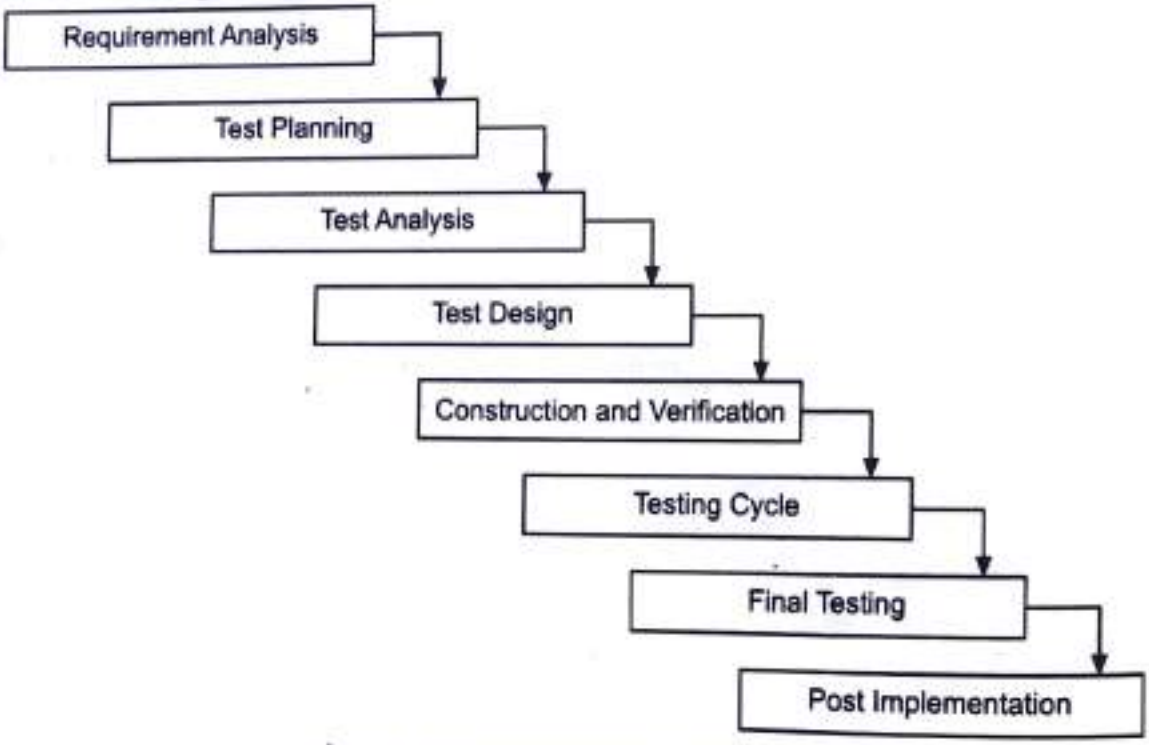


Fig. 1.8: Software Testing Life Cycle

Requirements Analysis:

In this phase, Software testers analyze the customer requirements and work with developers during the design phase to see which requirements are testable and how they are going to test those requirements.

It is very important to start testing activities from the requirements phase itself because the cost of fixing defects is very less if it is found in this phase rather than in future phases.

2. **Test Planning:**

- In this phase, all the planning about testing is done like what needs to be tested, how the testing will be done, test strategy to be followed, what will be the test environment, what test methodologies will be followed, hardware and software availability, resources, risks etc. A high level test plan document is created which includes all the planning inputs mentioned above and circulated to the stakeholders.

3. **Test Analysis:**

- After test planning phase is over, test analysis phase starts. In this phase, we need to dig deeper into project and figure out what testing needs to be carried out in each SDLC phase. Automation activities are also decided, if automation needs to be done for software products, how will the automation be done, how much time will it take to automate and which features need to be automated. Non functional testing areas (Stress and performance testing) are also analyzed and defined.

4. **Test Design:**

- In this phase, various Black-box and White-box test design techniques are used to design the test cases for testing. Testers start writing test cases by following those design techniques, if automation testing needs to be done then automation scripts also need to be written.

5. **Test Execution and Validation:**

- In this stage, we need to do all the test activities. All test cases designed in step 4 are verified and then processed with module(s) under test.

6. **Testing Cycle:**

- Test Execution and Bug Reporting, Bug Management.
- In this stage, we need to do test cycles until test cases are executed without errors and our test plans are reached as shown in Fig. 1.9.

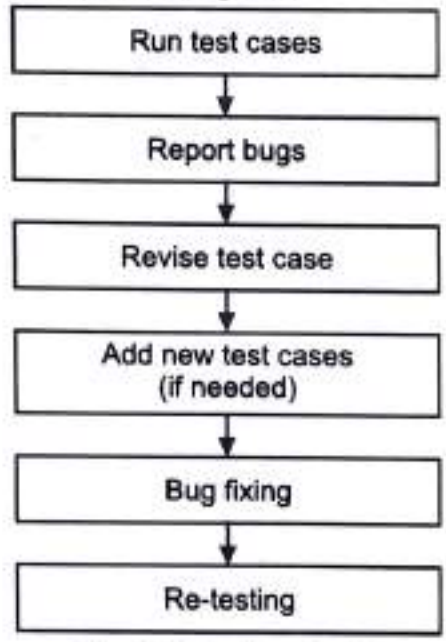


Fig. 1.9: Testing Cycle

7. **Final Testing and Implementation:**

- In this phase, the final testing is done for the software. Non functional testing like stress, load and performance testing are performed. Software is also tested in the production type of environment. In this phase, final test execution reports and all test deliverables are prepared.

8. **Post Implementation:**

- In this phase, the test environment is cleaned up and restored to default state, the process review meetings are done and lessons learnt are documented. A document is prepared to manage similar problems in future releases.

1.7.1 Types of Software Testing Methodologies

- In the field of Software testing, different types of Software Testing Methodologies are used. In the Software Development Process different software development approaches are used. A software development process also known as a Software Development Life Cycle (SDLC).
- Each process model has its own advantages and disadvantages, the choosing of the model is based on the requirement and need of the project to achieve the success in the process of software development.
- Here are the few commonly used Software Development Life Cycle used in actual testing:
  - Waterfall model
  - V model

1. **Waterfall Model:**

- The Waterfall Model approach is the most basic life cycle model. This is the first Process Model begins and used broadly in Software Development to ensure project success. This model was developed by Winston Royce in the early 1970. Waterfall model is software development ~~process~~ sequential process, as flowing progressively downwards like waterfall through multiple phases.
- The various phases involved in testing, with regard to the software development life cycle are:
  1. Requirements Analysis
  2. Test Planning
  3. Test Design
  4. Test implementation
  5. Test debugging
  6. System testing
  7. Acceptance testing
  8. Operations and maintenance

- Fig. 1.10 illustrates the Waterfall test process for software testing.

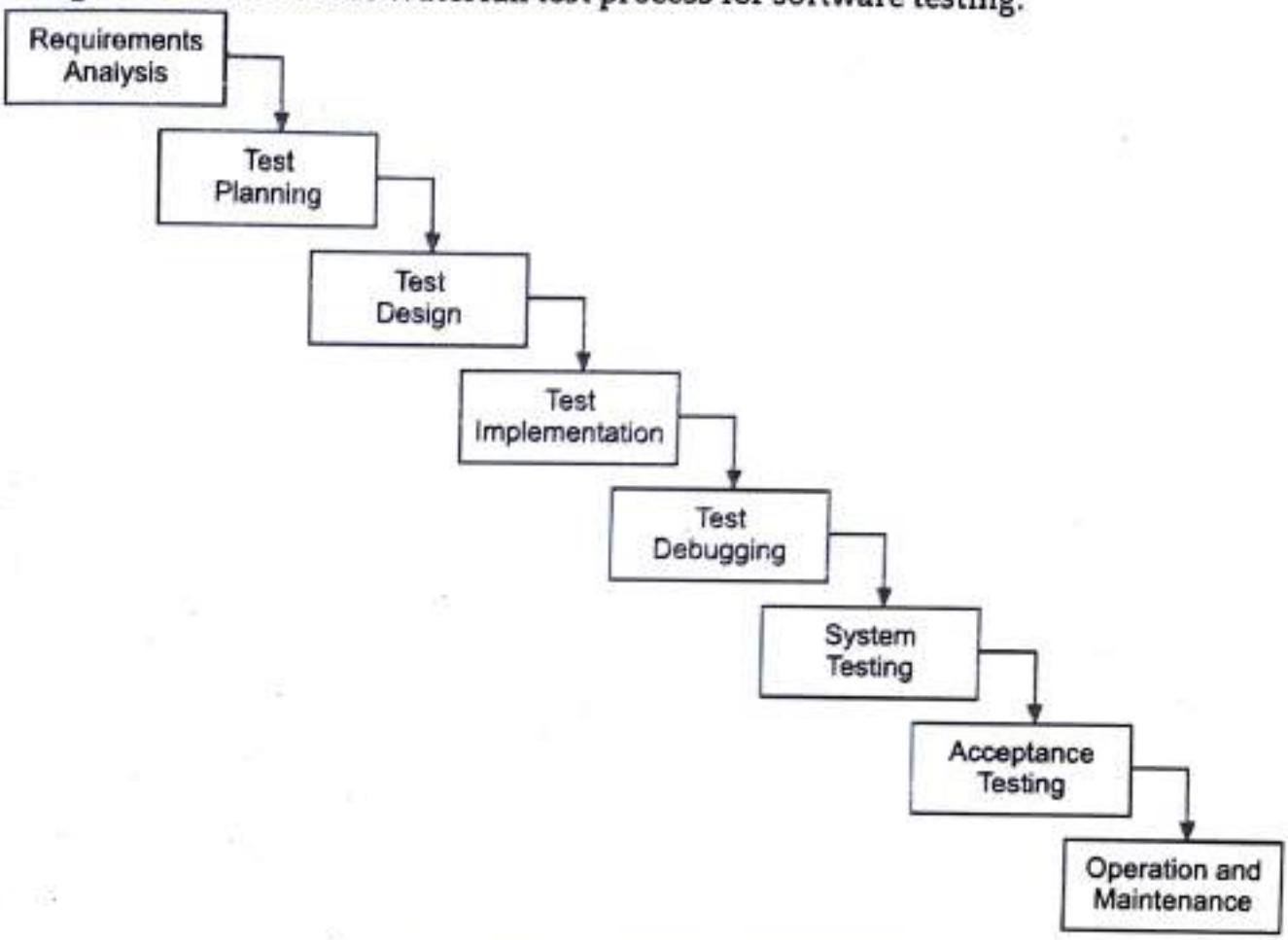


Fig. 1.10: Waterfall Model for Software Testing

(i) Requirements Analysis:

- When analyzing software requirements, the goals of the test team and the development team are somewhat different. Both teams need a clear, unambiguous requirements specification as input to their jobs. The development team wants a complete set of requirements that can be used to generate a system functional specification, and that will allow them to design and code the software. The test team, on the other hand, needs a set of requirements that will allow them to write a test plan, develop test cases, and run their system and acceptance tests.
- A very useful output of the requirements analysis phase for both development and test teams is a requirements traceability matrix. A requirements traceability matrix is a document that maps each requirement to other work products in the development process such as design components, software modules, test cases, and test results. It can be implemented in a spreadsheet, word processor table, database, or Web page.

(ii) Test Planning:

- A test plan document should be prepared after the requirements of the project are confirmed. Test planning includes determining the scope, approach, resources, and schedule of the intended testing activities. Efficient testing requires a substantial

investment in planning, and a willingness to revise the plan dynamically for changes in requirements, designs, or code, as bugs are uncovered.

- The requirements traceability matrix is a useful tool in the test-planning phase because it can be used to estimate the scope of testing needed to cover the essential requirements.
- Test planning phase should prepare the document for the system test and acceptance test phases that come near the end of the waterfall process, and should include:
  - Total number of features to be tested.
  - Mapping of tests to the requirements.
  - Testing approaches to be followed.
  - The testing methodologies.
  - Number of man-hours required.
  - Resources required for the whole testing process.
  - The testing tools that are to be used.
  - An assessment of test-related risks and a plan for their mitigation.

(iii) Test Design, Implementation, and Debugging:

- Dynamic testing relies on running a defined set of operations on a software build and comparing the actual results to the expected results. If the expected results are obtained, then the test counts as a pass; if inconsistent behavior is observed then test counts as a fail, but it may have succeeded in finding a bug. The set of operations that are defined to run, constitute a test case. Test cases need to be designed, written, and debugged before they can be used.
- A test design consists of two components: Test architecture and Detailed test designs.
  - The test architecture organizes the tests into groups such as functional tests, performance tests, security tests, and so on.
  - The detailed test designs describe the objective of each test, the equipment and data needed to conduct the test, the expected result for each test, and traces the test back to the requirement being validated by the test. Detailed test cases can be developed from the test designs. The level of detail needed for a written test procedure depends on the skill and knowledge of the people that run the tests.
- Test cases should be prepared based on the following scenarios:
  - Positive scenarios
  - Negative scenarios
  - Boundary conditions
  - Real World scenarios

- Once test cases are written, they are debugged by testing them against a build of the product software, which gives final test cases.

(iv) System Test:

- A set of finished, debugged test cases can be used in the next phase of the waterfall test process, system test. The purpose of system testing is to ensure that the software does what the customer expects it to do. There are two main types of system tests: function tests and performance tests.
- **Functional testing** requires no knowledge of the internal workings of the software, but it does require knowledge of the system's functional requirements. It consists of a set of tests that determines if the system does what it is supposed to do from the user's perspective. Once the basic functionality of a system is ensured, testing can turn to how well the system performs its functions.
- **Performance testing** consists of stress tests, volume tests, and recovery tests. Reliability, availability, and maintenance testing may also be included in performance testing.
- In addition to function and performance tests, there are a variety of additional tests that may need to be performed during the system test phase; these include security tests, install ability tests, compatibility tests, usability tests, and upgrade tests. If any failures occur, the bugs should be reported to the developer immediately to get the required workaround.

(v) Acceptance Test:

- When system testing is completed, the product can be sent to users for acceptance testing. If the users are internal to the company, the testing is usually called alpha testing. If the users are customers who are willing to work with the product before it is finished, the testing is beta testing. Both alpha and beta tests are a form of pilot tests in which the system is installed on an experimental basis for the purpose of finding bugs.
- When pilot testing is completed the customer gives feedback related to - which requirements are not satisfied or which requirements need to be changed in order to proceed to final testing.
- The final type of acceptance test is the installation test, which involves installing a completed version of the product at user sites for the purpose of obtaining customer agreement that the product meets all requirements and is ready for delivery.

(vi) Maintenance:

- Maintenance of a product is a challenging task for the development team and the test team. Maintenance for the developer consists of fixing bugs that are found during

customer operation and adding enhancements to product functionality to meet evolving customer requirements.

- For the test organization, maintenance means verifying bug fixes, testing enhanced functionality, and running regression tests on new releases of the product to ensure that previously working functionality has not been disturbed by the new changes.
- The testing process is an iterative process. Once the bugs are fixed, the testing has to be done repeatedly. Thus the testing process is an unending process.

2. **V Model of Testing :**

- The previous section has described waterfall models for the software development process and for the test process. The two models have common starting and ending points, but the test and development teams are involved in different activities.
- As we have seen in the Waterfall model, the issues found in the end of the SDLC. This is because the testing occurred at the end of SDLC. To overcome this problem the V-Model is comes into the picture. Here testing is done in the early phase and parallel to SDLC phases.
- Before starting the actual testing, the testing team has to work on various activities like preparation of Test Strategy, Test Planning, Creation of Test cases & Test Scripts which work parallel with the development activity which helps to get the test deliverable on time.

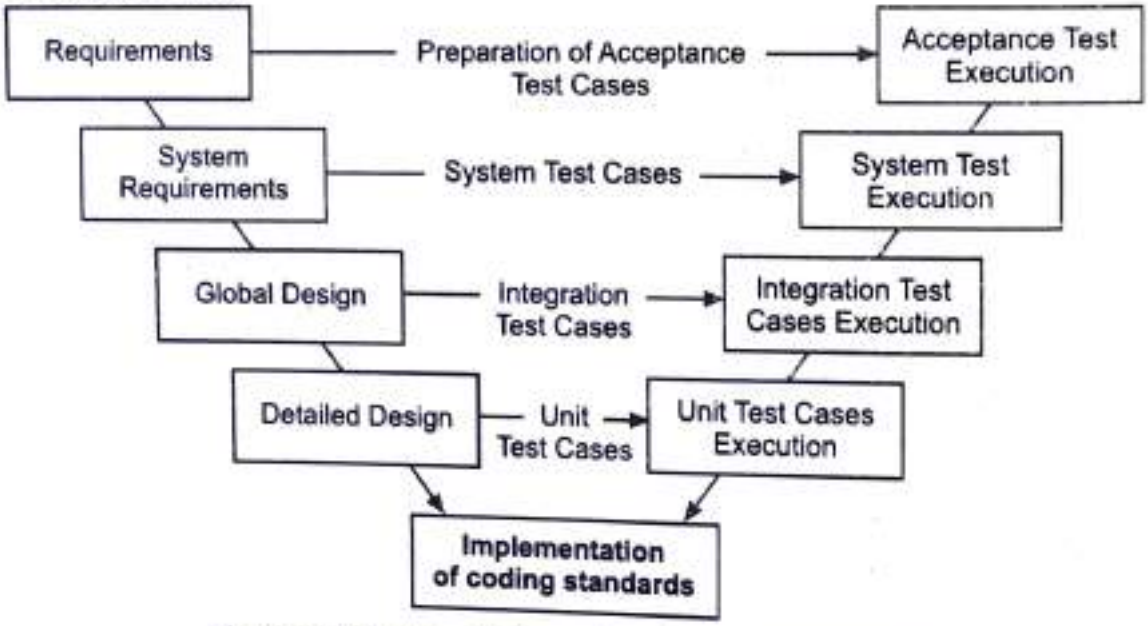


Fig. 1.11: V Model - Software Development Life Cycle

- In the V Model Software Development Life Cycle, based on basic information (Requirement Specification Document). Based on the requirement document, developer team starts working on the design and after completion on design, actual implementation starts and the testing team starts working on test planning, test case writing, test scripting. Both activities are working parallel to each other. In the

Waterfall model and-model they are quite similar to each other. Most of the organization is following this model because it is the most popular Software Testing Life Cycle model.

- The V-model is also called as *Verification and Validation model*. The testing activity is performed in each phase of the Software Testing Life Cycle phase. In the first half of the model, Verification testing activity is integrated in each phase like review user requirements, System Design document and in the next half the Validation testing activity occur.
- Typical V-model shows Software Development activities on the Left hand side of model and actual testing phases on the Right hand side of the model. In this process, "Do-Procedure" would be followed by the developer team and the "Check-Procedure" would be followed by the testing team to meet the mentioned requirements.
- In the V-Model Software Development Life Cycle, different steps are:
  1. **Unit Testing:** Preparation of Unit Test Cases.
  2. **Integration Testing:** Preparation of Integration Test Cases.
  3. **System Testing:** Preparation of System test cases.
  4. **Acceptance Testing:** Preparation of Acceptance Test Cases.

1.7.2 Debugging

[S-22, 23; W-22]

- Debugging is the activity which is carried out by the development team (or developer), after getting the test report from the testing team about defect(s) (defects can also be reported by the client). The developer then tries to find the causes of the defect. Developer may need to go through lines of code and find which part of code is having defects. After finding out the bug, he modifies that portion of code and then he rechecks if the defect is removed or not. After fixing the bug, the developer sends the software back to testers.

Guidelines:

- Some guidelines that are followed while performing debugging:
  1. Debugging is the process of solving a problem. Hence, before starting with debugging, members involved in debugging should understand all the causes of an error.
  2. While performing debugging, no experimentation should be done. The experimental changes instead of removing errors often increase the problem by adding new errors in it.
  3. When there is an error in one segment of a program, there is a high possibility that some other errors also exist in the program. Hence, if an error is found in one segment of a program, the complete program should be also examined for errors.

4. It should be ensured that the changed code added in the program to fix reported errors is correct and is not affecting the rest of the program. To verify such correctness and to ensure that no new errors are introduced, regression testing should be performed.

1.7.2.1 Phases of Debugging

- During debugging, errors are encountered that range from less damaging (like input of an incorrect function) to terrible (like system failure, which leads to economic or physical damage). Note that with an increase in the number of errors, the amount of effort to find their causes also increases.
- Once errors are identified in a software system, to debug the problem, a number of steps are followed, which are listed below:
  1. **Defect Identification:** A problem is identified in a system and a defect report is created.
  2. **Defect Confirmation:** A software engineer maintains and analyzes this error report and finds solutions to the following questions before confirming the defect.
     - Does a defect exist in the system?
     - Can the defect be reproduced?
     - What is the expected/desired behavior of the system?
     - What is the actual behavior?
     - Is it a genuine defect?
  3. **Defect Analysis:** If the defect is real, the next step is to understand the root cause of the problem. Generally, engineers debug by starting a debugging tool (debugger) and they try to understand the root cause of the problem by following a step-by-step execution of the program. Here, defect is analyzed to understand its severity.
  4. **Defect Resolution:** Once the root cause of a problem is identified, the error can be resolved by making an appropriate change to the code by fixing the problem.
- When the debugging process ends, the software is retested to ensure that no errors are left undetected. Moreover, it checks that no new errors are introduced in the software while making some changes to it during the debugging process.

1.7.2.2 Debugging Strategies

- As debugging is a difficult and time-consuming task, it is essential to develop a proper debugging strategy. This strategy helps in performing the process of debugging easily and efficiently. The commonly-used debugging strategies are debugging by brute force, induction strategy, deduction strategy, backtracking strategy, and debugging by testing.

1. **Brute force method of debugging:**

- This is the most commonly used but least efficient method. It is generally used when all other available methods fail. Here, debugging is done by taking memory (or storage) dumps. Actually, program is loaded with output statements that produce a large amount of information including the intermediate values. Analyzing this information may help to identify the error cause. However, using a memory dump for finding errors requires analyzing huge amounts of information or irrelevant data. This is a time consuming process and requires a lot of effort and skill.

2. **Induction Strategy:**

- This strategy is a 'disciplined thought process' where errors can be debugged by moving outwards from the particulars to the whole. This strategy assumes that once the symptoms of the errors are identified, and the relationships between them are established, then the errors can be detected by observing these symptoms and their relationships. To perform an induction strategy, a number of steps are followed, which are listed below:

  (a) **Locating relevant data:** All the information about a program is collected to identify the functions, which are executed correctly and incorrectly.

  (b) **Organizing data:** The collected data is organized according to importance. The data can consist of possible symptoms of errors, their location in the program, the time at which the symptoms occur during the execution of the program and the effect of these symptoms on the program.

  (c) **Devising hypothesis:** The relationships among the symptoms are studied and a hypothesis is devised that provide hints about the possible causes of errors.

  (d) **Proving hypothesis:** In the final step, the hypothesis needs to be proved. It is done by comparing the data in the hypothesis with the original data to ensure that the hypothesis explains the existence of hints completely. In case, the hypothesis is unable to explain the existence of hints, it is either incomplete or contains multiple errors in it.

3. **Deduction Strategy:**

- In this strategy, first step is to identify all the possible causes. Then each cause is analyzed and if found invalid, the cause is removed. Deduction strategy is also based on some assumptions.
- To use this strategy, the following steps are involved.

  1. **Identifying the possible causes**: A list of all the possible causes of errors is formed. Using this list, the available data can be structured and analyzed.
  2. **Eliminating possible causes using the data:** The list is examined to recognize the most probable cause of errors and least probable causes are deleted.

3. **Refining the hypothesis:** By analyzing the possible causes one by one and lookin for contradiction leads to elimination of more invalid causes. This results in refined hypothesis containing few specific possible causes.
4. **Proving the hypothesis:** This step is similar to the fourth step in inductio method.

4. **Backtracking Strategy:**

- This method is effectively used for locating errors in small programs. According t this strategy, when an error has occurred, one needs to start tracing the progra backward one step at a time evaluating the values of all variables until the cause o error is found. This strategy is not useful with a large program as the number o backward paths increases and will become a tedious job.

5. **Debugging by Testing:**

- This debugging method can be used in conjunction with debugging by induction an debugging by deduction methods. Additional test cases are designed that help i obtaining information to devise and prove a hypothesis in the induction method an to eliminate the invalid causes and refine the hypothesis in the deduction metho Note that the test cases used in debugging are different from the test cases used i the testing process. Here, the test cases are specifically designed to explore th internal program state.

1.7.3 Test Plant

- Test plan is a document describing the scope, approach, resources, and schedule o intended testing activities.
- The test plan acts as the anchor for the execution, tracking, and reporting of th entire testing project. Project test plan describes the overall strategy that the proje will follow, for testing the final application. The role of a test plan is to guide al testing activities.
- Test plan for a software projects is a very detailed document containing followin items:

1. **Overall test objectives.** The test plan should clearly define test objectives. Fo example, why are we testing, what is to be achieved by the tests, and what are th risks associated with testing?
2. **What is to test (scope of the tests)?** What items, features, procedures, function objects and subsystem 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 techniqu are going to be applied?

5. **When to test?** The test plan should show how the stages of the testing process, such as component, integration and acceptance, correspond to stages of the development process. Testing can begin as soon as some coherent unit is developed and continues on successively larger units until the complete application is tested.
6. **Risks** that may be faced with appropriate improvement plans.

1.7.3.1 Hierarchy of Test Plan

- Test plans can be organized in several ways depending on organizational policy. There is a hierarchy of plans that includes several levels of quality assurance and test plans. A sample plan hierarchy is shown in Fig. 1.12.

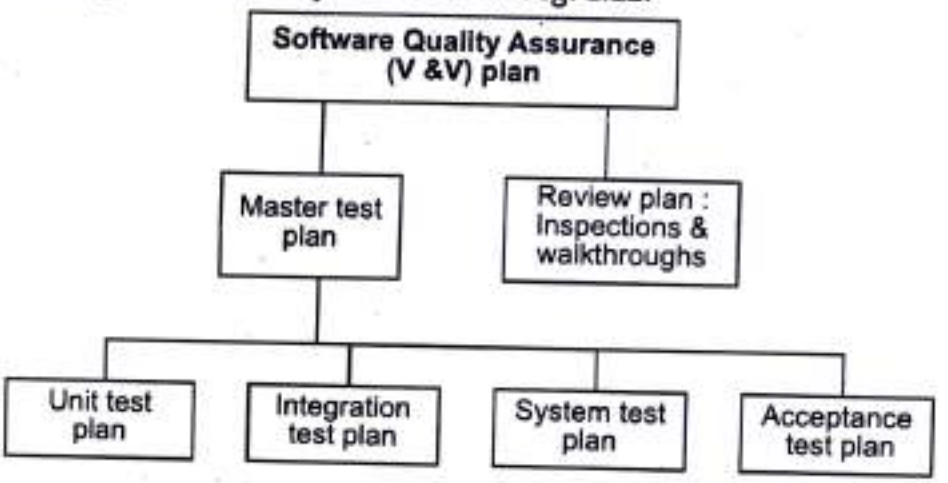


Fig. 1.12: A Hierarchy of Test Plan

- **Software Quality Assurance plan** gives an overview of all verification and validation activities for the project. Master test plan includes an overall description of all execution-based testing for the software system. Depending on organizational policy, another level of hierarchy could contain a separate test plan for unit, integration, system and acceptance tests. In some organizations, they are included in the master test plan.

1.7.3.2 Difference between the Test Plan and Test Strategy

- Test plan differs from Test strategy. Table 1.2 shows the difference between the test plan and strategy.

Table 1.2: Test Plan Vs Test Strategy

Test plan	Test strategy
"A software project test plan is a document that describes the objectives, scope, approach, and focus of a software testing effort".	Test Strategy or Test Approach is a set of guidelines that describes the test process.

Contd...

Test plan covers – What needs to be tested, How testing is going to be performed? Resources needed for testing, Timelines and Risk associated.	Test strategy covers - How testing is going to be performed? What will be the test architecture?
Test Plan document is a high level document and normally developed by the Project Manager.	The Test Strategy document is usually prepared by the Test Lead or Test Manager.
Test plan is the set of ideas that guide a test project.	Test Strategy is the set of ideas that guide test design.
Test plan includes test strategy.	Test strategy is a subset of test plan.

1.7.3.3 Test Plan Components

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.

2. **Introduction:**

- Gives an overall description of the project:
  - Software system being developed or maintained.
  - Test objective and scope (scope covers both what is in the scope and what is not).
- That is, deciding which features will be tested and which will not be.
- For testing, scope management also involves – prioritizing the features for testing. The following factors drive the choice and prioritization of features to be tested:
  - Features that is new and critical for the release.
  - Features whose failures can be catastrophic.
  - Features that are expected to be complex to test.
  - Features which are extensions of earlier features that have been defect prone.

3. **Items to be tested:**

- This is a listing of the entities to be tested and should include names, identifiers, and version number for each entity. The items listed could include procedures, classes, modules, subsystems, and systems. References to the appropriate documents where these items are described such as requirements and design documents and the user manual should be included in this component of the test plan.

4. **Test Environment:**

- It describes the test environment including hardware, software, and operating system.

- A testing environment is a set-up of software and hardware on which the testing team is going to perform the testing of newly built software products.
- This set-up consists of the physical set-up which includes hardware, and logical set-up that includes Server Operating system, Client operating system, Database server, Front end running environment, Browser (if web application), IIS (version on server side) or any other software components required to test this software product.
- This testing set-up is to be built on both the ends i.e. the server and client.

5. **Test team:**

- List testing participants and their role. For example, two test engineers, John and Sally will be responsible for test case design, test execution, and recording of the results. Robert - test manager, and Jane - test lead engineer, will supervise them.

6. **Testing Strategies:**

This includes identifying:

(a) What type of testing would be used for testing the functionality?

(b) What are the configurations or scenarios for testing the features?

(c) What "non-functional" tests would you need to do?

- Commonly used testing strategies are: Manual Testing (Black box testing and/or White box testing) or Automated Testing (Winrunner, SilkTest, Load runner etc.).

7. **Test Deliverables:**

- Execution based testing has a set of deliverables:
  - **Test approach:** This explains the objectives and scope of the test.
  - **Test Scenario:** Provide high-level descriptions of functionality to be tested. It may include various scenarios and sub-scenarios based on the project requirements.
  - **Test scripts (test cases):** This provide step-by-step instructions and detailed results based on test scenarios.
  - **Test summary report, and defect report.**

8. **Test Schedule:**

- A test schedule lists the activities, tasks, or events that make-up a test plan. Test Schedule section of the test plan includes - schedules for each milestone or iteration, each with its own start and end date. One can also list key project dates, such as the date of the final release, code freeze, UI freeze, beta entry, beta exit, and so on.

For example, the project should be ready for acceptance test by "dd/mm/yyyy". Schedules for use of tools and equipment should also be specified.

Table 1.3 : Sample Test Schedule

Test activities	Start date	End date	Person days	Resource
Prepare test plan				
Review test plan				
Executing test cycle1				
Unit testing				
Integration testing				
System testing				
Customer/Acceptance testing				

9. Exit Criteria:

- Include criteria for when to stop testing. Test plan should specify the conditions to suspend testing based on the effects or severity level of the failures/defects observed.
- A decision to stop testing will be based on tracking:
  (1) Coverage of all the requirements.
  (2) The number of open defects reported.

10. Risk Management:

- Every testing effort has risks associated with it. These risks should be:
  1. Identified.
  2. Evaluated in terms of their probability of occurrence.
  3. Prioritized.
  4. Emergency plans should be developed that can be activated if risk occurs.
- Some of the risks are enlisted below:
  1. **Delays in testing efforts.** If the testing schedule is significantly impacted by high severity level defects, an additional developer will be assigned to the project to perform fault localization.
  2. Application itself is not available for the testing on time.
  3. All the resources are not properly provided.
  4. SRS is not clearly defined.

11. Defect Recording/Tracking:

- A tool such as spreadsheet should be developed for recording defects and tracking status. Defects can be tracked using various tools such as Bugzero, Bugzilla, Bug tracker software etc.

12. Testing costs:

- Test costs that should be included following things in the plan are:
  - Costs of planning and designing the tests.
  - Costs of acquiring the hardware and software necessary for the tests.
  - Costs to support the test environment.
  - Costs of executing the tests; recording and analyzing test results.
  - Teardown costs to restore the environment.

13. Approvals:

- The test plans for a project should be reviewed by those designated by the organization. All parties who review the plan and approve it, should sign the document.

1.7.3.4 Test Plan Review

- Review is a filter for the software engineering process. It is a group meeting whose purpose is to evaluate a product or set of products and relevant information.
- Test plan review should ensure that all the components of the test plan are present and that they are correct, clear, and complete. The general document checklist can be applied to test plans. An example of the test plan checklist is shown in table 1.3.

Table 1.4: A sample checklist for a Test Plan Review

Test Plan Review
Test Items: • Are all items to be tested included in the test plan? • Has each requirement, feature and design element been covered in the test plan? • Has the testing approach been clearly described? • Have all the test deliverables been included?
Test Environment: • Has the test environment been described clearly, and does it include hardware and software needs etc. • Has time for setting and tearing down the environment been allocated?
Testing Risks: • Have all the risks associated with testing the software product been identified and analyzed in the test plan?
Testing Costs: • Does the plan account for testing costs? • Are testing costs compatible with those specified in the project plan?

1.7.3.5 Difference between Testing and Debugging

- Software testing should not be confused with debugging. Debugging is the process of analyzing and locating bugs when software does not behave as expected.
- Debugging is therefore an activity which supports testing, but cannot replace testing. However, no amount of testing can be guaranteed to discover all bugs.

Summary

- Testing is an intrinsic part of the entire software development lifecycle. Though it occurs after the coding phase in traditional SDLC, but its need arises from the requirement gathering phase itself.
- In this chapter, we have discussed the basic testing principles, testing objectives and testing life cycle. The occurrences of defects and its frequency can be controlled effectively if testing is implemented in parallel with the development life cycle in terms of verification and validation. We have learned the Review concept and various types of reviews.

Check Your Understanding

1. Basic goal of software testing is _____.
   (a) to find the defect.
   (b) to review the software.
   (c) to ensure that software is as per the specifications.
   (d) All of the above
2. Which of the following are Post implemented goals of Software Testing?
   (a) Reduce Maintenance Cost
   (b) Bug detection
   (c) Bug Recovery
   (d) Bug prevention
3. The system is in the state such that further processing by the system will lead to failure.
   (a) Error
   (b) Failure
   (c) Faults
   (d) Defects
4. Which of the following is not a part of the review?
   (a) Design Review
   (b) Code Review
   (c) Requirement review
   (d) Implementation review
5. Which statement is NOT correct?
   (a) Software team is a part of the Walkthrough process.
   (b) Review is Static testing.
   (c) Inspection is an informal review in presence of all team.
   (d) Review can be formal.

6. _____is a process of confirming that software meets the specifications.
   (a) Verification (b) Validation
   (c) Debugging (d) Quality control
7. Dynamic testing is _____.
   (a) Review type (b) Inspection type
   (c) Walkthrough type (d) None of the above
8. Which of the following is a component of the test plan?
   (a) Test plan Identifier (b) Test Analysis
   (c) Final testing (d) All of the above
9. V and V is called as _____.
   (a) Verification and Validation (b) Verification and Verification
   (c) Validation and Validation (d) Verification with V loop
10. Functional testing is _____.
   (a) Verification technique (b) Review technique
   (c) Validation technique (d) Coding technique

ANSWERS

1. (d)	2. (a)	3. (a)	4. (d)	5. (c)	6. (a)	7. (d)	8. (d)	9. (a)	10. (c)

Practice Questions

Q. I Answer the following questions in short.

1. What is Software Testing?
2. Explain terms – Error, Fault and Failure.
3. What is Informal Review?
4. Why should we test software?
5. What is static testing?
6. What is debugging?
7. What is the purpose of Software testing?

Q. II Answer the following questions.

1. What is the difference between Functional and Non-functional testing?
2. Which are the principles of testing?
3. Describe the need of Software Testing?
4. What is debugging? Explain with its phases.
5. What is a Debugging Strategy?
6. Explain Waterfall Testing Cycle with diagram.

7. Which professionals are involved in Software Testing?
8. Write the difference between Inspection and Reviews.

Q. III Define the terms.

1. FTR
2. STLC
3. Waterfall model
4. Walkthrough
5. Code Review

2... Approaches to Testing – Testing Methods

Learning Objectives ...

- To study of White Box Testing and its types.
- To study of Black Box Testing and its types.
- To learn about Test Case Design.
- To know about Gray Box Testing.

2.1 INTRODUCTION

- We already know a test is an activity in which a system is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system.
- Software testing is a process of evaluating a product and its conformance. There are two types of testing i.e. Static and Dynamic.
- Software testing is a critical element of software quality assurance and represents the ultimate process to ensure the correctness of the product.

Static and Dynamic Testing:

Static Testing: [W-22; S-23]

- Static testing is a non-execution based technique i.e. document-based testing. It involves reviewing, understanding the requirements, coding, and design documents. Each document undergoes reviews against checklists, standards, and practices.
- Static testing includes Informal Review, Walkthrough and Inspections.

Dynamic Testing:

- Dynamic testing is an execution-based testing where the actual test execution takes place by the independent testers or developers. Entire system will undergo a testing process to check if the actual and expected results are meeting or not.

Dynamic Testing is divided into two types i.e. Black box and White box which are explained in next Sections.

Table 2.1 Difference between Static and Dynamic Testing [W-22; S-23]

Sr. No.	Static Testing	Dynamic Testing
1.	Static Testing done without executing the program.	Dynamic Testing done by executing the program.
2.	This testing does verification process.	Dynamic testing does the validation process.
3.	Static testing is about prevention of defects.	Dynamic testing is about finding the defects.
4.	Static testing gives assessment of code and documentation.	Dynamic testing gives bugs/ bottlenecks in the software system.
5.	Static testing involves a checklist and process to be followed.	Dynamic testing involves test cases for execution.
6.	This testing can be performed before execution.	Dynamic testing is performed during execution.
7.	Cost of finding defects and fixing them is low.	Cost of finding and fixing defects is high.
8.	Return on investment will be high as this process is involved at early stage.	Return on investment will be low as this process involved after the coding phase.
9.	More reviews and comments are highly recommended for good quality.	Early defects are highly recommended for good quality.
10.	Requires loads of meetings.	Comparatively requires lesser meetings.

2.2 WHITE BOX TESTING

[W-22; S-23]

- White Box Testing (WBT) is also known as Code-Based Testing or Structural Testing.
- Some synonyms of white box testing are:
  1. Glass box testing.
  2. Clear box testing.
  3. Open box testing.
  4. Transparent box testing.

5. Structural testing.
6. Logic driven testing.
7. Design based testing.

- White box testing is the software testing method in which internal structure is known as the tester who is going to test the software.
- White box testing involves testing by looking at the internal structure of the code and when you are completely aware of the internal structure of the code then you can run your test cases and check whether the system meets requirements mentioned in the specification document. Based on derived test cases the user exercised the test cases by giving the input to the system and checking for expected outputs with actual output. In this testing method the user has to go beyond the user interface to find the correctness of the system.
- For testers to test the software application under test is like a white/transparent box where the inside of the box is clearly seen to the tester (as tester is aware/access of the internal structure of the code), so this method is called as White Box Testing.
- The White box testing is one of the best methods to find out the errors in the software application in early stage of software development life cycle. In this process the deriving the test cases is most important part. The test case design strategy includes such that all lines of the source code will be executed at least once or all available functions are executed to complete 100% code coverage of testing.

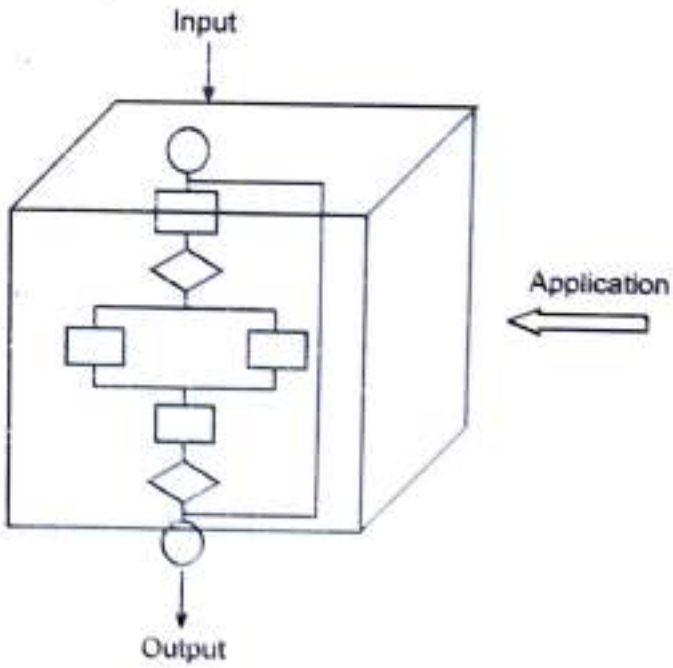


Fig. 2.1: White Box Testing

- In White Box Testing, the tester has access to the program's code and can examine it for clues to help him with his testing.

[S-22]

White Box Testing can be defined as,

Testing based on design and implementation structures (specification may be ignored). It deals with the internal logic and structure of the code. It is testing from the inside - tests that go in and test the actual program structure, which provides information about flow of control. For example: if-then-else statements, loops, etc.

- The tests written based on the white box testing strategy incorporate coverage of the code written, branches, paths, statements and internal logic. It will not test missing functions (i.e. those described in the functional design specification but not implemented by the internal specification or code).
- It is good at catching following errors in implementation:
  - Initialization errors.
  - Bounds-checking and out-of-range problems.
  - Buffer overflows errors etc.

Advantages of White Box Testing:

- To start the testing of the software no need to wait for the GUI, you can start the White Box Testing.
- As covering all possible paths of code so this is a detailed testing.
- Tester can ask about implementation of each section, so it might be possible to remove unused lines of code which might be causing the introduction of bugs.
- By executing equivalence use to approximate the partitioning.
- As the tester is aware of internal coding structure, then it is helpful to derive which type of input data is needed to test software applications effectively.
- White box testing allows you to help in the code optimization.

Disadvantages of White Box Testing:

- To test the software application a highly skilled resource is required to carry out testing who knows the deep knowledge of internal structure of the code which increases the cost.
- Update test script is required if changing the implementation too frequently.
- If the application under test is large then exhaustive testing is impossible.
- It is not possible for testing each and every path/condition of a software program, which might miss the defects in code.
- White box testing is a very expensive type of testing.
- To analyze each line by line or path by path is nearly impossible work which may introduce or miss the defects in the code.

Table 2.2: Difference between Black Box and White Box testing

Sr. No.	Black box testing (Functional Testing)	White box testing (Means Structural Testing)
1.	**Focus: I/O behavior**. For any given input, predict the output, if it matches with actual output then the module passes the test.	**Focus: Thoroughness (coverage)**. Every statement in the component is executed at least once.
2.	Tester does not have access to program code. He only knows what the software is supposed to do. He doesn't know HOW or WHY it happens?	Tester has access to the program's code and can examine it for clues to help him with his testing.
3.	It is testing against functional specification.	Testing based on design and implementation structures (specification may be ignored).
4.	It is not based on any knowledge of internal design or code. Tests are based on requirements and functionality. Therefore, programming knowledge is not required.	It deals with the internal logic and structure of the code.
5.	It will not test hidden functions and errors associated with them will not be found.	It will not test missing functions (i.e. those described in the functional design specification but not implemented by the internal specification or code).
6.	Black box testing is so named as it is not possible to look inside the box to determine the design and implementation structures of the components you are testing.	White box testing is so named as it is possible to look inside the box to determine the design and implementation structures of the components you are testing.
7.	**Synonyms for Black box:** Behavioral testing, Functional testing, Opaque-box testing, Concrete Box testing and Closed-Box testing.	**Synonyms for White box:** Structural testing, Clear-Box testing, Glass-Box testing, Open-Box testing.

8.	**Black box Testing Types :** 1. Static Black Box Testing. 2. Dynamic Black Box Testing.	**White box Testing Types:** 1. Static White Box Testing. 2. Dynamic White Box Testing. 3. Statement Coverage. 4. Branch Coverage. 5. Path Coverage.

2.2.1 Working process of White Box Testing

- White box testing is focusing on the structure and flow of the system. It is used for verifying the security holes in the code, incomplete paths in the code, Verify the flow of structure mentioned in the specification document, Verify the Expected outputs Verify all conditional loops in the code.
- It verifies the line by line or Section by Section flow in the code and then ensures 100% coverage.
- The pre-designed test cases are executed with the help of input data and compare the output with the expected one and find a mismatch if any means you found a bug.

2.2.2 Types of White Box Testing

- There are different types and different methods for each white box testing type that is shown in the following figure.

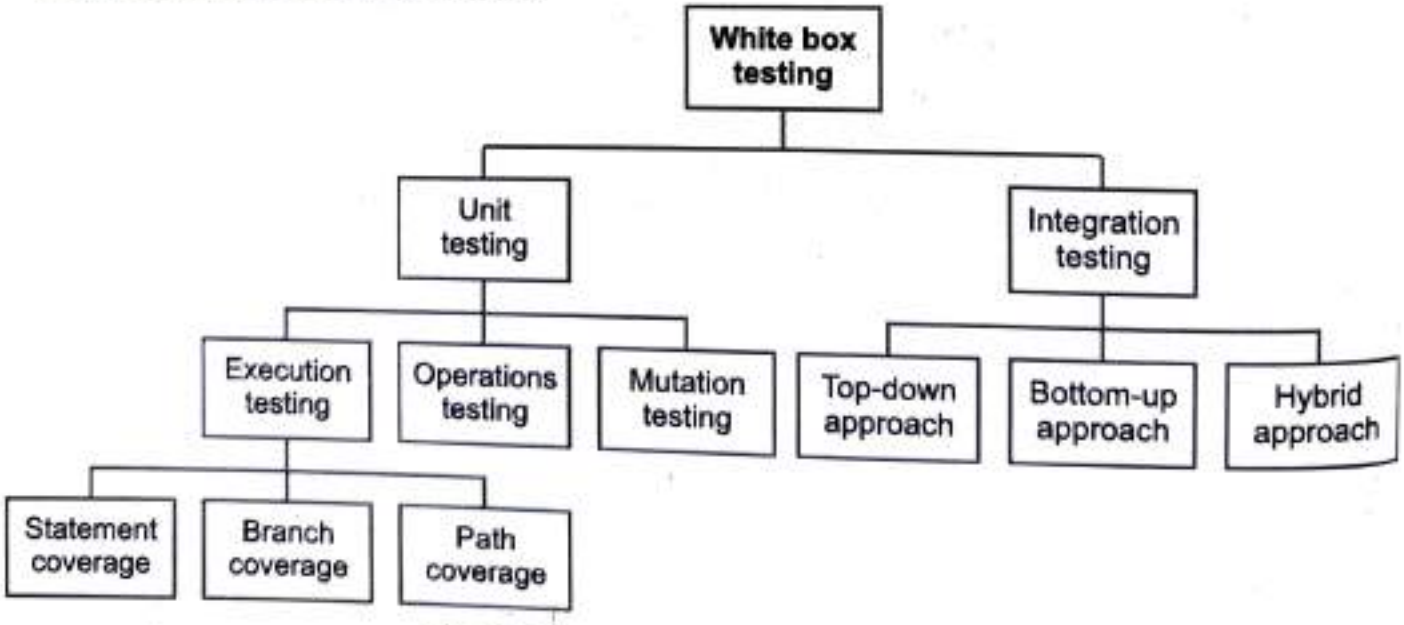


Fig. 2.2: Types of White box testing

2.2.2.1 Unit Testing

- The developer carries out unit testing in order to check if the particular module or unit of code is working fine. Unit Testing comes at the very basic level as it is carried out as and when the unit of the code is developed or a particular functionality is built

- Unit test is the most basic testing mechanism at the developer level. This covers a very narrow and well defined scope. We isolate the code from any outside interaction or any dependency on any module. Unit tests focus on very small units of functionality. They cover the interaction of the code with memory only and do not cover any interaction with network, database or file systems. These dependencies are hard coded into the code while testing.
- They provide a simple way to check the smallest units of code and prove that units can work perfectly in isolation. However, we need to check further that when these units are combined they work in a cohesive manner which leads us to further types of tests.

1. Execution testing:

- In the White box testing the main part is the Code Coverage analysis which helps to identify the gaps in a test case. It allows you to find the area in the code to which is not executed by a given set of test cases. Upon identifying the gap in the test case suite you can add the respective test case. So it helps to improve the quality of the software application. The adequacy of the test cases is often measured with a metric called coverage. Coverage is a measure of the completeness of the set of test cases.
  - (a) **Segment/Statement Coverage**: Ensure that each code statement is executed once.
  - (b) **Branch Coverage or Node Testing**: Coverage of each code branch in from all possible ways.
  - (c) **Compound Condition Coverage:** For multiple conditions, test each condition with multiple paths and combination of different paths to reach that condition.
  - (d) **Basis Path Testing (Path Coverage)**: Each independent path in the code is taken for testing.
- The main White box testing Techniques are:

I. Statement Coverage:

- In programming language, a statement is nothing but the line of code or instruction for the computer to understand and act accordingly. A statement becomes an executable statement when it gets compiled and converted into the object code and performs the action when the program is in running mode. Hence "Statement Coverage", as the name suggests, is the method of validating that each and every line of code is executed at least once.
- The Statement Coverage is also known as Line Coverage or Segment Coverage. The Statement Coverage covers only the true conditions. Through statement coverage we can identify the statements executed and where the code is not executed because of blockage. In this process each and every line of code needs to be checked and executed.

- The Statement Coverage can be calculated as shown below:

$$\textbf{Statement Coverage} = \frac{\textbf{Number of statements exercised}}{\textbf{Total number of statements}} \times 100\%$$

Example:

- If you are monitoring statement coverage while you test your software, your goal is t make sure that you execute every statement in the program at least once.
- In the case of the short program shown in Listing, 100 percent statement coverag would be the execution of lines 1 through 4.

Listing:

```
1: PRINT "Hello World"
2: PRINT "The date is: "; Date$
3: PRINT "The time is: "; Time$
4: END
```

- You might think this would be the perfect way to make sure that you tested you program completely. You could run your tests and add test cases until ever statement in the program is touched.
- Unfortunately, statement coverage is misleading. It can tell you if every statement i executed, but it can't tell you if you have taken all the paths through the software.

Advantage of Statement Coverage:

(i) It verifies what the written code is expected to do and not to do.

(ii) It measures the quality of code written.

(iii) It checks the flow of different paths in the program.

Disadvantage of Statement Coverage:

(i) It cannot test the false conditions.

(ii) It does not report whether the loop reaches its termination condition.

(iii) It does not understand the logical operators.

- **Tools:** To test the Statement Coverage the Cantata++ can be used as white box testin tool.

II. Branch Coverage:

- Attempting to cover all the paths in the software is called Path testing. The simples form of path testing is called Branch Coverage Testing. The goal of branch coverage i to ensure that whenever a program can jump, it jumps to all possible destinations.
- Branch coverage is a testing method, which aims to ensure that each one of th possible branches from each decision point is executed at least once and thereb ensuring that all reachable code is executed.
- "Branch" in programming language is like the "IF statements". If a statement has tw branches: true and false. So in Branch coverage (also called Decision coverage), w validate that each branch is executed at least once.

- In case of a "IF statement", there will be two test conditions:
  1. To validate the true branch.
  2. To validate the false branch.
- Hence in theory, Branch Coverage is a testing method which when executed ensures that each branch from each decision point is executed.
- The branch coverage testing can be calculated as shown below:

$$\textbf{Branch coverage Testing} = \left(\frac{\textbf{Number of decisions outcomes tested}}{\textbf{Total number of decision outcomes}}\right) \times 100\%$$

Example 1:

```
Read A
Read B
IF A+B > 10 THEN
   Print "A+B is large"
ENDIF
If A > 5 THEN
   Print "A is large"
ENDIF
```

- The above logic can be represented by a flowchart as shown in Fig. 2.3.

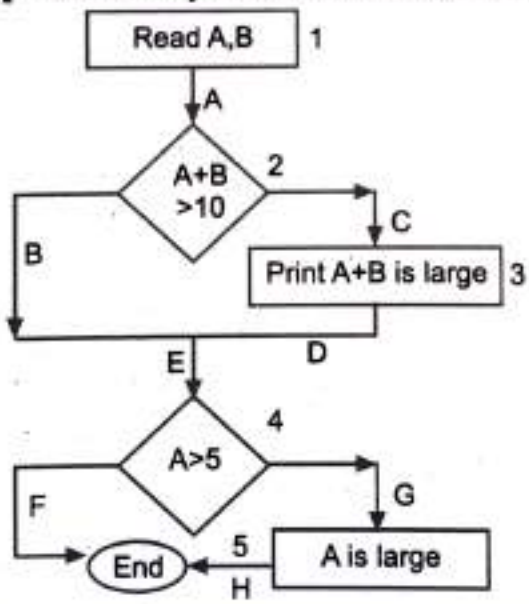


Fig. 2.3: Flowchart for Branch Coverage

Result:

- To calculate Branch Coverage, one has to find out the minimum number of paths which will ensure that all the edges are covered. In this case there is no single path which will ensure coverage of all the edges at once. The aim is to cover all possible true/false decisions. In this example there are two conditions with four branches and four total decision outcomes which can be covered with following flows:
  1. 1A-2C-3D-E-4G-5H

2. 1A-2B-E-4F-5H

Hence Branch Coverage is 2.

ample 2:

Consider the following program.

Program 2.1: The IF statement creates another branch through the code:

```
1: PRINT "Hello World"
2: IF Date$ = "01-01-2021" THEN
3: PRINT "Happy New Year"
4: END IF
5: PRINT "The date is: "; Date$
6: PRINT "The time is: "; Time$
7: END
```

- If you test this program with the goal of 100 percent statement coverage, you would need to run only a single test case with the Date$ variable set to January 1, 2021.
- The program would then execute the following path:

  Lines 1, 2, 3, 4, 5, 6, 7
- Your code coverage analyzer would state that you tested every statement and achieved 100 percent coverage. You could quit testing. You may have tested every statement, but you didn't test every branch.
- You still need to try a test case for a date that's not January 1, 2021. If so, the program would execute the other path through the program:

  Lines 1, 2, 5, 6, 7
- Most code coverage analyzers will account for code branches and report both statement coverage and branch coverage results separately giving you a much better idea of your test's effectiveness.

Tools:

- To cover the Decision Coverage testing in the code the **TCAT-PATH** is used. This supports the C, C++ and Java applications.

III. Compound Condition Coverage:

- Condition Coverage is a measure of percentage of Boolean sub-expressions of the program that have been evaluated as both true or false outcome [applies to compound predicate] in test cases. So Condition Coverage is also known as Predicate Coverage.
- The Condition Coverage testing can be calculated as shown below:

$$\text{Condition Coverage} = \frac{\text{Total decision exercised}}{\text{Total number of decisions}} \times 100\%$$

- Above program shows a slight variation. An extra condition is added to the IF statement in line 2 that checks the time as well as the date.

- Condition Coverage testing takes the extra conditions on the branch statements into account.

Program 2.2: The multiple conditions in the IF Statement creates more paths through the code.

```
1: PRINT "Hello World"
2: IF Date$ = "01-01-20 21" AND Time$ = "00:00:00" THEN
3: PRINT "Happy New Year"
4: END IF
5: PRINT "The date is: "; Date$
6: PRINT "The time is: "; Time$
7: END
```

- In this sample program, to have full condition coverage testing, you need to have the four sets of test cases shown in Table 2.3. These cases assure that each possibility in the IF statement is covered.

Table 2.3: Test Cases to achieve Full Condition Coverage

Date$	Time$	Line # Execution
00-00-2021	00:00:00	1,2,5,6,7
01-01-2021	11:11:11	1,2,5,6,7
00-00-2021	11:11:11	1,2,5,6,7
01-01-2021	00:00:00	1,2,3,4,5,6,7

- If you were concerned only with branch coverage, the first three conditions would be redundant and could be equivalence partitioned into a single test case. But, with condition coverage testing, all four cases are important because they exercise the different conditions of the IF statement in line 2 False-True, True-False, False-False, and True-True. As with branch coverage, code coverage analyzers can be configured to consider conditions when reporting their results. If you test for condition coverage, you will achieve branch coverage and therefore achieve statement coverage.

IV. Path Coverage:

- Path Coverage tests all the paths of the program. This is a comprehensive technique which ensures that all the paths of the program are traversed at least once. Path Coverage is even more powerful than Branch Coverage.
- This technique is useful for testing complex programs. In the actual development process developers make use of the combination of techniques that are suitable for the software application.

In this coverage, we split a computer program into a number of distinct paths. A program can start from the beginning and take any of the paths to its completion.

Path Coverage calculated as shown below:

$$\text{Path Coverage} = \frac{\text{Total path exercised}}{\text{Total number of paths in program}} \times 100\%$$

- Using above mentioned white box testing techniques, the 80% to 90% code coverage is completed.

2. **Operational Acceptance Testing (OAT):**

- Operational Acceptance is used to conduct operational readiness (pre-release) of a product, service or system as part of a quality management system. OAT is a common type of non-functional software testing, used mainly in software development and software maintenance projects.
- This type of testing focuses on the operational readiness of the system to be supported, and/or to become part of the production environment. Hence, it is also known as Operational Readiness Testing (ORT) or Operations Readiness and Assurance (OR and A) testing. Functional testing within OAT is limited to those tests which are required to verify the non-functional aspects of the system.

3. **Mutation Testing:**

- In this type of testing, the application is tested for the code that was modified after fixing a particular bug/defect. It also helps in finding out which code and which strategy of coding can help in developing the functionality effectively.
- Besides all the testing types given above, there are some more types which fall under both Black box and White box testing strategies such as: Functional testing (which deals with the code in order to check its functional performance), Incremental integration testing (which deals with the testing of newly added code in the application), Performance and Load testing (which helps in finding out how the particular code manages resources and give performance etc.).

2.2.2.2 Integration Testing

- When individual software modules are merged and tested as a group then it is known as Integration Testing. Integration testing is sets between Unit Testing and System Testing.
- Integration testing is executed to establish whether the components interact with each other consort to the specification or not. Integration testing in large refers to by the developer or the software tester or by both. Example - checking that a Payroll system interacts as required with the Human Resource system.

Integration Testing Example:

- If you have to test the keyboard of a computer then it is a unit testing but when you have to combine the keyboard and mouse of a computer together to see its working or not than it is the integration testing. So it is prerequisite that for performing integration testing a system must be unit tested before.
- Black box test case design tactics are the most typical during integration, although a limited amount of testing of white box may be used to ensure description of major control paths.
- Integration testing is always sub-divided as follows:

Types of Integration Testing:

- Techniques of integration testing are given below:
  1. Top-down testing approach.
  2. Bottom-up testing approach.
  3. Big-Bang testing approach.
  4. Sandwich testing approach.

  (** For more information, see section 3.3).

2.2.3 How to perform White Box Testing?

- Let's take a simple website application. The end user simply accesses the website, Login and Logout. From the end user's point of view, users are able to access the website from GUI but inside there are lots of things going on. To check the internal things, going right or not, the white box testing method is used. To explain this we have to divide this part in two steps. This is all being done when the tester is testing the application using White box testing techniques.

  Step 1: Understand the Source Code.

  Step 2: Create Test Cases and Execute.

Step 1: Understand of the Source Code:

- The first important step is to understand the source code of the application being tested. The tester should know about the internal structure of the code which helps to test the application. Better knowledge of source code will help to identify and write the important test case which causes the security issues and also helps to cover the 100% test coverage. Before doing this the tester should know the programming language that is being used in developing the application.
- As security of application is the primary objective of application so testers should be aware of the security concerns of the project which help in testing. If the tester is alert of the security issues then he can prevents the security issues like attacks from hackers and users who are trying to insert the malicious things in the application.

Step 2: Create Test Cases and Execute:

- The second step involves the actual writing of test cases based on Statement/Decision/Condition/Branch coverage and actual execution of test cases to cover the 100% testing coverage of the software application. The tester will write the test cases by dividing the applications by Statement/Decision/Condition/Branch wise. Other than this tester can include the trial and error testing, manual testing and software testing tools.
- To test each path or condition may require different input conditions, so to test full application tester needs to create fill range of inputs which may be time consuming.

2.3 TEST CASE DESIGN [S-22, 23; W-22]

- Test cases are input values used for testing a module. Test cases are designed using White box and Black box test case designing technique.

2.3.1 Designing of White Box Test Case

- It is a useful test case design technique for structural testing.
- It uses the internal structure of software to derive test cases.
- Logical paths are checked using specific sets of conditions and/or loops.
- Here system design is taken into consideration.

Use of test case:

- White box testing method generate test cases to:
  - To ensure that all independent paths within a module have been checked at least once.
  - To ensure that all logical decisions are exercised on their true as well as false side.
  - To ensure that all loops along with their boundaries are exercised within their operational bounds.
  - To ensure the validity of all types of internal data structures.

Types:

- Different white box test case design techniques are:
  1. Basis Path Testing:
     - Draw the flow graph.
     - Calculate the Cyclomatic complexity.
  2. Coverage Testing:
     - Statement Coverage-Code and Data Coverage.
     - Branch and Decision Coverage.

2.3.1.1 Basis Path Testing

- Basis Path Testing is a white box testing technique. It is based on the control structure of a program or a module.

- It enables the logical complexity measure of a procedural design, which can be used to define a basis set of execution paths.
- It gives all possible paths for which the test case needs to be defined.
- This technique guarantees that every statement in a program is executed at least once during testing.
- Designs flow graph andmeasures the complexity by calculating the Cyclomatic complexity of process.
  - In flow graph notation, each circle represents a Node. Node is representing the procedural statement.
  - Flow graph notation terminology uses nodes, arrows, regions and Predicate nodes.
    - Node represents process, arrow represents links or edges, and Region is the area bounded by edges and nodes.
    - Each node that contains a condition i.e. logical OR, AND etc. is called a Predicate Node.
    - Predicate node always has one incoming edge and two outgoing edges.

Procedure of Basis Path Testing:

- Steps involved in Basis Path Testing are:

1. Design the Flow graph:
   - Read the algorithmic code.
   - Identify nodes and predicate nodes.
   - Design the flow graph by connecting nodes with edges.
2. Find the Cyclomatic complexity:
   - Cyclomatic complexity is a quantitative way for measuring logical complexity. It is used to identify the independent set of paths for which test cases have to be designed. By checking those paths it can guarantee that the complete code is tested in such a way that all the nodes are tested at least once. This way defines independent paths in the basis set of the program.
   - The value of Cyclomatic complexity provides an upper bound for the number of tests conducted to ensure that all the statements of code are executed at least once.
   - Cyclomatic Complexity is computed in one of three ways :
     1. Cyclomatic complexity V(G) = Number of regions R
     2. Cyclomatic complexity V(G) = No. of Predicate nodes P + 1
     3. Cyclomatic complexity V(G) = E − N + 2

        Where, E = Number of Edges and N = Number of Nodes.

- Some structured constructs in flow graph are:

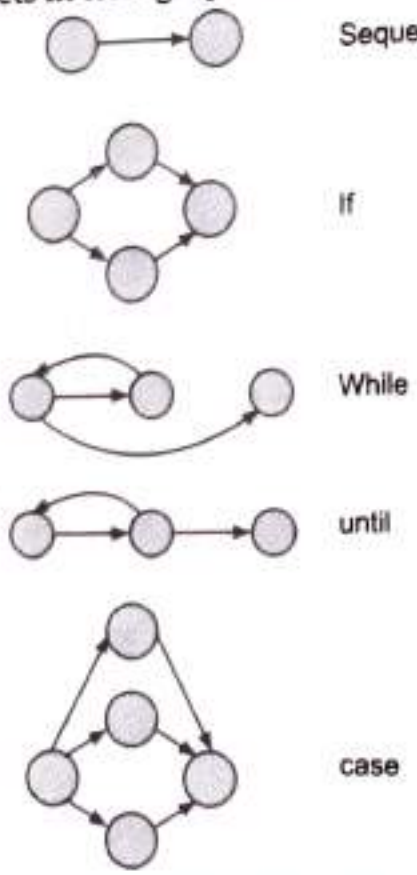


Fig. 2.4: Tool-based Static Techniques

- Here, each circle represents one or more non-branching source code statements. Each arrow represents an outgoing edge. When more than one arrow/edge falls in to one node then it is called as a junction node. When there exists more than one outgoing edge from a node then that node is called as a branch node or predicate node.
- Cyclomatic Complexity provides the quantitative measure of the logical complexity of a program. In the basis path testing context, the complexity value defines the number of independent paths in a program. This provides a limit for the maximum number of test cases that must be conducted to assure that all statements have been executed at least once. An independent path is any path through the program that introduces at least new set of processing statements or a new condition. In a flow graph, an independent path must move along at least one such edge which has not been traversed before the path is defined.

Example:

- Consider an algorithm to read a file record by record and should process the record by storing them in a buffer if the record is not zero otherwise it should increment the counter. If record 2 is also zero then the code should reset the counter otherwise it should process the record. Calculate the Cylomatic complexity for the defined code and drive the test cases for linearly independent paths.

Solution:

- Consider the following Procedure algorithm:

```
1: do while records remain
2: read record;
3: if record field 1!= 0
4: then process record
   store in buffer;
5: increment counter;
6: elseif record field 2 = 0
7: then reset counter;
8: else process record
   store in file;
9: endif
10: enddo
11: end
```

Step 1: Draw the FlowChart

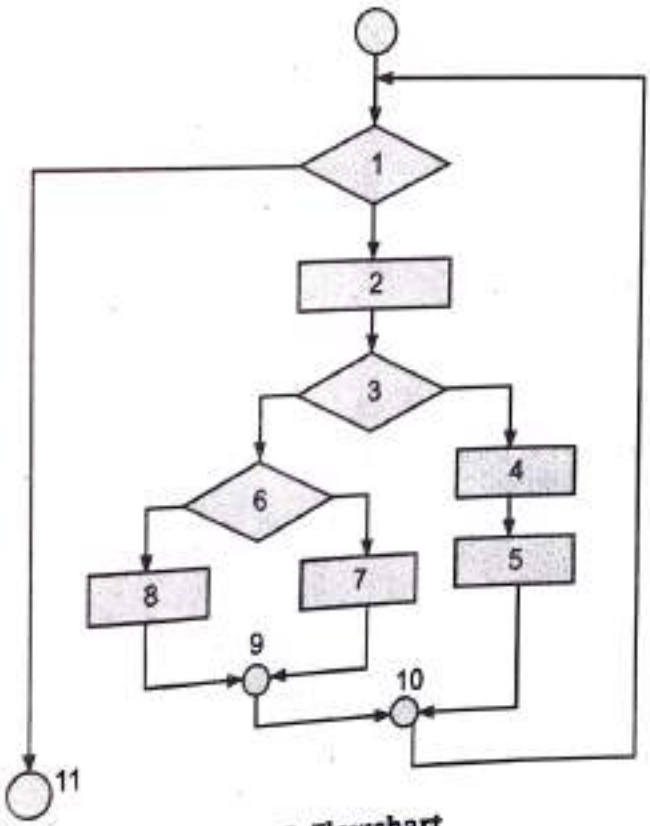


Fig. 2.5: Flowchart

Step 2: Give Single node to sequence statements

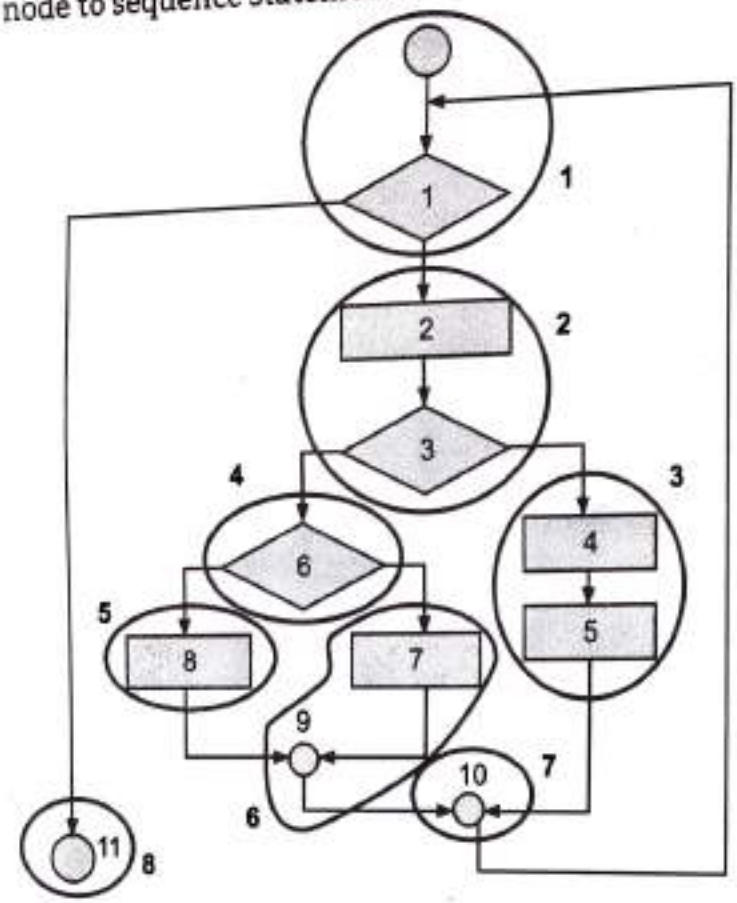
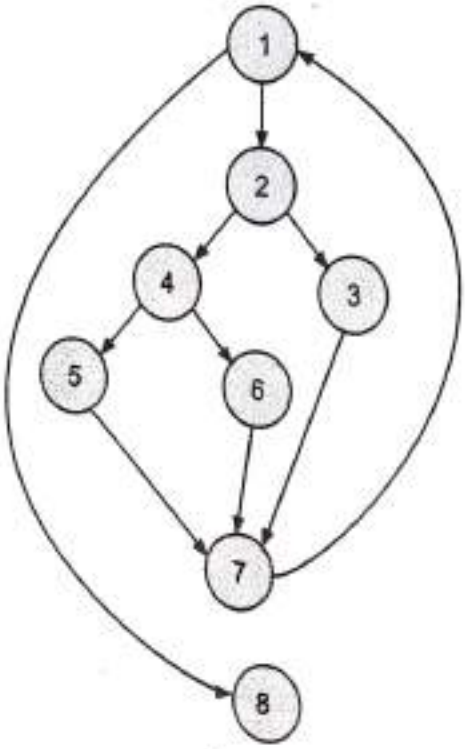
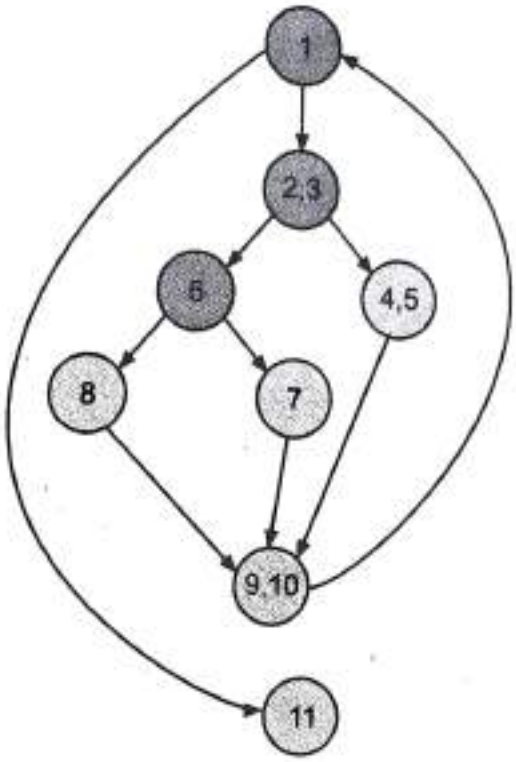


Fig. 2.6: Single node to Sequence statements

Step 3: Draw the flow graph



(a)



No. of predicate Nodes : = 3
No. of Nodes = 8
No. of Edges = 10
So Cyclometric Complexity =
No. of edges-nodes + 2 = 10 – 8 + 2 = 4
No. of Predicate Nodes +1 = 3 + 1 = 4

(b) Flow graph

Fig. 2.7

Path 1 : 1 – 11
Path 2 : 1-2-3-4-5-9-10-1-11
Path 3 : 1-2-3-6-8-9-10-1-11
Path 4 : 1-2-3-6-7-9-10-1-11

th 1: 1 – 11

- **Test case:** Check for record value = 0;
- **Expected Result:** With record value = 0 program should reach to end.

th 2: 1-2-3-4-5-9-10-1-11

- **Test case:** check for record field 1 = 0
- **Expected Result:** Record will be processed and stored and counter is incremented.

th 3: 1-2-3-6-8-9-10-1-11

- **Test case:**
  (a) Check for record field 1 != 0
  (b) Check for record field2 =0
- **Expected Result:** Record field2 will be processed and stored and counter is incremented.

th 4: 1-2-3-6-7-9-10-1-11

- **Test case:** a) Check for record field 2 != 0
- **Expected Result:** Reset counter and go for next iteration.

2.3.1.2 Coverage Testing Technique

- Coverage means how much of the structure has been exercised or covered by a set of tests.
- Coverage techniques are used for two purposes: Test measurements and Test case design.
- They are often used to assess the amount of testing done.
- They are used to design additional tests to increase test coverage.

Disadvantages:

1. Coverage techniques measure only one dimension of a multi-dimension concept.
2. Coverage techniques measure the coverage of software code that has been written. They cannot test the software that has not been written.
3. 100% coverage does not mean 100% tested.

How to measure test coverage?

- Test coverage can be measured based on a number of different structural elements in software. Data coverage includes data flow. It involves checking of data in any form. It ensures that every form of data is checked for its defined standard.
  - Tracking a piece of data completely through the software.
  - Check the values at various places.
  - Check for the overflow errors. Like variables are declared but the values assigned after calculations are beyond the limits.
  - Involve checking of boundaries and sub-boundaries. For example if your software accepts a range of 1 to 1000 with the boundary conditions- 0,1,2 (for boundary 1) and 999,1000,1001 (for boundary 1000) then it also should include the other sub-boundary values if any is there.
  - Suppose if you are testing a text-box that accepts only the characters a-z and A-Z, then you should include in your invalid partition the values just below and above those specified in the ASCII table like:- -,@,[,',{
  - For checking formulae and equations you should check for the zero value in the denominator. It may be directly entered or may be the result of another computation. At the time of checking the equations like: `A = P[1+r/n]exp(nt)` then a good tester would consider the condition n=0 and then checks whether the value of n is generated internally or by other functions.
- It supports error forcing and provides usability checking. It checks for the desired error messages to be appearing on screen to prompt the user whenever it applies. Here, purposely such conditions are generated that will result in the errors or failures of the software. Here, you check whether that message gives proper direction to the user or not i.e. check for usability.

- Code coverage includes checking for the coding in the program. A code comprises varieties of statements, like simple statement, conditional statement and branch statement. It is discussed in more detail inside statement coverage and branch coverage in following sections.

1. **Statement Coverage:**

- (Note: We have discussed Statement coverage in detail in Section 2.2.)

Example:

2.3.2 Black Box Test Case Designing

- Black box testing is a test case design technique used to check the functionality of a code. It is a functional oriented test case design technique to check the code by giving certain data as input and by observing the produced output to ensure that it is performing as per the expected defined behavior.
- Following are the Black box test case designing techniques (covered in details in the section 2.4)
  1. Equivalence Partitioning.
  2. Boundary Value Analysis.
  3. Comparison Testing.

2.4 BLACK BOX TESTING [S-22]

- Black Box Testing, also known as Behavioral Testing, is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester. These tests can be functional or non-functional, though usually functional.
- Functional testing is concerned with what the system does with its features or functions. Non-functional testing is concerned with examining how well the system does. Non-functional testing includes performance, usability, portability, maintainability, etc.
- This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see. This method attempts to find errors in the following categories:
  - Incorrect or missing functions.
  - Interface errors.
  - Errors in data structures or external database access.
  - Behavior or performance errors.
  - Initialization and termination errors.

Fig. 2.8: Black Box Testing

Definition by ISTQB:

- **Black Box testing:** Testing, either functional or non-functional, without reference to the internal structure of the component or system.
- **Black Box test design technique:** Procedure to drive and/or select test cases based on an analysis of the specification, either functional or non-functional, of a component or system without reference to its internal structure.

- Black Box (or functional) testing checks the functional requirements and examines the input and output data of these requirements. Once the specified function for which the software has been designed is known, tests are performed to ensure that each function is working properly. This is referred to as black box testing.
- In this testing, the tester only knows what the software is supposed to do, he/she cannot look in the box to see how it operates. If he/she types in a certain input, he/she gets a certain output. He/She doesn't know how or why it happens, just that it does.
- Black box testing focuses on the functional requirements of the software. That is, black box testing techniques enable you to drive sets of input conditions that will fully exercise all functional requirements for a program. For Example: A tester, without knowledge of the internal structures of a website, tests the web pages by using a browser; providing inputs (clicks, keystrokes) and verifying the outputs against the expected outcome.
- Black box testing helps in the overall functionality verification of the system under test.

1. **Black box testing is done based on requirements:** Black box testing helps in identifying any incomplete, inconsistent requirement as well as any issues involved when the system is tested as a complete entity.
2. **Black box testing addresses the stated requirements as well as implied requirements:** Not all the requirements are stated explicitly, buy are deemed implicit.
3. **Black box testing encompasses the end user perspectives:** Since, we want to test the behavior of a product from an external perspective, end-user perspectives are an integral part of black box testing.
4. **Black box testing handles valid and invalid inputs:** Black box testing is natural for users to make errors while using a product. Hence, it is not sufficient for black box testing to simply handle valid inputs. Testing from the end-user perspective includes testing for these errors or invalid conditions. These are called positive and negative test cases.

The software tester may not know the technology or the internal logic of the product. However, knowing the technology and the system internals help in constructing test cases specific to the error-prone areas.

Since, requirements specifications are the major inputs for black box testing, test design can be started early in the cycle.

Levels Applicable To:

- Black Box testing method is applicable to the following levels of software testing:
  - Integration Testing
  - System Testing
  - Acceptance Testing
- Higher the level, and hence the bigger and more complex the box, the more black box testing methods come into use.

2.4.1 Black Box Testing Techniques [S-22]

- There are various techniques to be used for doing black box testing. These various techniques are shown in Fig. 2.5.

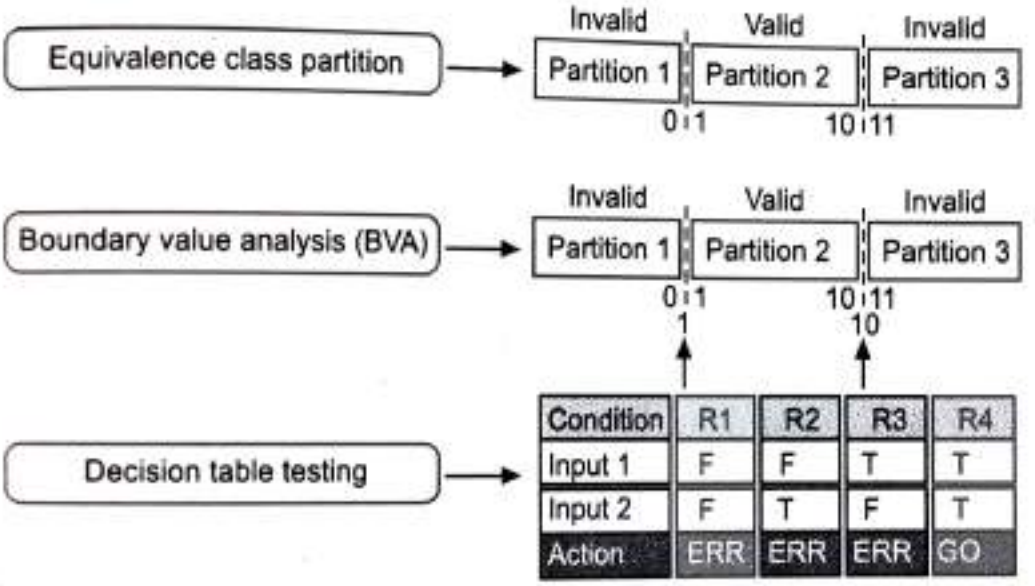


Fig. 2.9: Techniques of Black Box testing

- Following are some techniques that can be used for designing black box tests.

1. **Equivalence Partitioning:**

- Equivalence Partitioning (EP) is a specification-based or black box testing technique. This technique is very common and mostly used by all the testers informally. Equivalence partitions are also known as Equivalence Class Partition (ECP). It divides the input domain of a program into classes of data from which test cases can be derived. The goal of equivalence partitioning is to reduce the set of possible test cases into a smaller, manageable set that still adequately tests the software.
- Equivalence partitioning is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data. It can be applied at any level of testing and is often a good technique to use first.
- The idea behind this technique is to divide (i.e. to partition) a set of test conditions into groups or sets that can be considered the same (i.e. the system should handle them equivalently), hence 'equivalence partitioning'. Equivalence partitions are also known as equivalence classes – the two terms means exactly the same thing.

- In the equivalence-partitioning technique we need to test only one condition from each partition. This is because we are assuming that all the conditions in one partition will be treated in the same way by the software. If one condition in a partition works, we assume all of the conditions in that partition will work, and so there is little point in testing any of these others. Similarly, if one of the conditions in a partition does not work, then we assume that none of the conditions in that partition will work so again there is little point in testing any more in that partition.
- This is a software testing technique to minimize the number of permutations and combinations of input data. In equivalence partitioning, data is selected in such a way that it gives as many different output as possible with the minimal set of data.
- If software behaves in an identical way for a set of values, then the set is termed as an equivalence class or a partition. It can be assumed safely that functionality of the software will be the same for any data value from the equivalence class or partition. Input data is analyzed and divided into equivalence classes which produce different output.
- So essentially, if you want to use equivalence partitioning in your projects, there are two steps that you need to follow:
  - Identifying equivalence classes or partition.
  - Picking one value from each partition for the complete coverage.
- Main advantage of using this technique is that testing efforts are minimized and at the same time coverage is also ensured. Using this technique, redundancy of test cases is removed by eliminating data which does not produce different output.

Example:

- Consider a very simple function for awarding grades to the students. This program follow the given guideline to award grades:
  - Marks 00 - 39 ---------- Grade D
  - Marks 40 - 59 ---------- Grade C
  - Marks 60 - 70 ---------- Grade B
  - Marks 71 - 100 ---------- Grade A
- Based on the equivalence partitioning techniques, partitions for this program could be as follows :
  - Marks between 0 to 39 - Valid Input
  - Marks between 40 to 59 - Valid Input
  - Marks between 60 to 70 - Valid Input
  - Marks between 71 to 100 - Valid Input
  - Marks less than 0 - Invalid Input
  - Marks more than 100 - Invalid Input
  - Non numeric input - Invalid Input

- From the above example, it is clear that from infinite possible test cases (Any value between 0 to 100, Infinite values for > 100 < 0 and non numeric) data can be divided into seven distinct classes. Now even if you take only one data value from these partitions, your coverage will be good.

- A filename in windows OS can contain any characters except w : * ? " <> and |. Filenames can have from 1 to 255 characters. If you're creating test cases for filenames, you will have equivalence partitions for valid characters, invalid characters, valid length names, names that are too short, and names that are too long.

2. Boundary Values Analysis (BVA): [W-22; S-23]

- Boundary Value Analysis is a software test design technique that involves determination of boundaries for input values. It selects values that are at the boundaries and just inside/outside of the boundaries as test data.
- Boundary Value Analysis (BVA) is based on testing at the boundaries between partitions. Here we have both valid boundaries (in the valid partitions) and invalid boundaries (in the invalid partitions). A boundary value is any input or output value on the edge of an equivalence partition.
- Suppose you have a very important tool at the office, accept valid User Name and Password fields to work on that tool, and accept minimum 8 characters and maximum 12 characters. Valid range 8-12, Invalid range 7 or less than 7 and Invalid range 13 or more than 13.

Example:

- Write Test Cases for Valid partition value, Invalid partition value and exact boundary value.

  Test Cases 1: Consider password length less than 8.

  Test Cases 2: Consider a password of length exactly 8.

  Test Cases 3: Consider passwords of length between 9 and 11.

  Test Cases 4: Consider a password of length exactly 12.

  Test Cases 5: Consider passwords of length more than 12.

- The best way to describe boundary condition testing is explained by the following example.
- If software can operate on the edge of its capabilities, it will almost certainly operate well under normal conditions. Boundary conditions are special because Programming, by its nature, is vulnerable to problems at its edges.
- The following code shows how a boundary condition problem can make its way into a very simple program.

BASIC Program Demonstrating a Boundary Condition Bug:

```
1:  Rem Create a 10 element integer array
2:  Rem Initialize each element to -1
3:  Dim  data(10)  As  Integer
```

```
4: Dim i As Integer
5: For i = 1 To 10
6: data (i) = -1
7: Next i
8: End
```

- The purpose of this code is to create an array of 10 elements and initialize each element of the array to -1. An array data of 10 integers i.e. data (10) and a counter i are created. A for loop runs from 1 to 10, and each element of the array from 1 to 10 is assigned a value of -1.
- In most BASIC scripts, when an array is dimensioned with a stated range in this case, `Dim data (10) as Integer` - the first element created is 0, not 1. This program actually creates a data array of 11 elements from data (0) to data (10). The program loops from 1 to 10 and initializes those values of the array to -1. But the first element of the array is data(0), it does not get initialized.
- When the program completes, the array values look like this:

  data (0) = 0
  data (1) = -1
  data (2) = -1
  data (3) = -1
  data (4) = -1
  data (5) = -1
  data (6) = -1
  data (7) = -1
  data (8) = -1
  data (9) = -1
  data (10) = -1
- Notice that data (0)'s value is 0, not -1. If the same programmer later forgot about, or a different programmer wasn't aware of how this data array is initialized, he might use the first element of the array, data (0), thinking it was set to 1. Problems such as this are very common and, in large complex software, can result in very horrible bugs.

Types of Boundary Conditions:

- Boundary conditions are those situations at the edge of the planned operational limits of the software.
- When you're presented with a software test problem that involves identifying boundaries, look for the following types:
  - Numeric
  - Character
  - Position

- Quantity
- Speed
- Location
- Size

- Following are the boundary conditions that need to be tested:
  - First/Last.
  - Min/Max.
  - Start/Finish.
  - Empty/Full.
  - Shortest/Longest.
  - Slowest/Fastest.
  - Largest/Smallest.
  - Highest/Lowest.
- Testing outside the boundary is usually as simple as adding one, or a bit more, to the maximum value and subtracting one, or a bit more, from the minimum value. For example:

```
1. First1/Last+1
2. Start1/Finish+1
3. Less than Empty/More than Full
4. Largest+1/Smallest1
5. Min1/Max+1
```

Examples:

1. If a text entry field allows 1 to 255 characters, try entering 1 character and 255 characters as the valid partition. You might also try 254 characters as a valid choice. Enter 0 and 256 characters as the invalid partitions.
2. If a program reads and writes to a CD-R, try saving a file that's very small, may be with one entry. Save a file that's very large just at the limit for what the disc holds. Also try saving an empty file and a file that's too large to fit on the disc.
3. If a program allows you to print multiple pages onto a single page, try printing just one (the standard case) and try printing the most pages that it allows. If you can, try printing zero pages and one more than it allows.
4. The software may be has a data-entry field for a 9-digit ZIP code. Try entering 00000-0000, the simplest and smallest. Try entering 99999-9999 as the largest. Try entering one more or one less digit than what's allowed.
5. If you are testing a flight simulator, try flying right at ground level and at the maximum allowed height for your plane. Try flying below ground level and below sea level as well as into outer space.

6. Consider a printer that has an input option of the number of copies to be made, from 1 to 99. To apply boundary value analysis, we will take the minimum and maximum (boundary) values from the valid partition (1 and 99 in this case) together with the first or last value respectively in each of the invalid partitions adjacent to the valid partition (0 and 100 in this case). In this example, we would have three equivalence partitioning tests (one from each of the three partitions and four boundary value tests.

3. Use of Decision Table in Testing:

- A decision table is a good way to deal with different combination inputs with their associated outputs and also called cause-effect table. Reason to call a **cause-effect table** is an associated logical diagramming technique called cause-effect graphing that is basically used to derive the decision table.
- We can apply Equivalence Partitioning and Boundary Value Analysis techniques to only specific conditions or inputs. Although, if we have dissimilar inputs that result in different actions being taken or secondly we have a business rule to test that there are different combinations of inputs which result in different actions.
- We use a decision table to test these kinds of rules or logic. It is a tool that helps us look at the "complete" combination of conditions. Decision tables are very much helpful in test design technique. It helps testers to search the effects of combinations of different inputs and other software states that must correctly implement business rules. They also provide a regular way of stating complex business rules, that's helpful for developers as well as for testers.
- Testing combinations can be a challenge, as the number of combinations can often be huge. It assists in the development process with developers to do a better job. Testing with all combinations might be unrealistic or unfeasible. It is basically an outstanding technique used in both testing and requirements management. It is a structured exercise to prepare requirements when dealing with complex business rules. Also, used in model complicated logic.
- The conditions in the decision table may take on any number of values. When it is binary, then the decision table conditions are just like a truth table set of conditions. The decision table allows the iteration of all the combinations of values of the condition, thus it provides a "completeness check".
- The conditions in the decision table may be interpreted as the inputs, and the actions may be thought of as outputs.

Cause Effect Graph (Decision table):

- It is a software test design technique that involves identifying the cases (input conditions) and effects (output conditions), producing a Cause-Effect Graph, and generating test cases accordingly.

Use of decision tables for test designing:

- The first task is to identify a suitable function or subsystem which reacts according to a combination of inputs or events. The system should not contain too many inputs otherwise the number of combinations will become unmanageable. It is better to deal with large numbers of conditions by dividing them into subsets and dealing with the subsets one at a time. Once you have identified the aspects that need to be combined, then you put them into a table listing all the combinations of True and False for each of the aspects.

Examples:

1. **Consider the scenario of Printer Troubleshooting:**

Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognized	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

2. **Employees working on salaried or hourly basis:**

	Conditions/ Course of Action	Rules					
		1	2	3	4	5	6
Condition Stubs	Employee type	S	H	S	H	S	H
	Hours worked	<40	<40	40	40	>40	>40
Action Stubs	Pay base salary	X		X		X	
	Calculate hourly wage		X		X		X
	Calculate overtime						X
	Produce Absence Report		X				

Advantages of Decision-Table:

1. Allow us to start with a "complete" view, with no consideration of dependence.
2. Allow us to look at and consider "dependence", "impossible" and "not relevant" situations and eliminate some test cases.
3. Allow us to detect potential errors in our specifications.

Disadvantages of Decision-Table:

1. Need to decide (or know) what conditions are relevant for testing.
2. Scaling up can be massive: 2n for n conditions.

4. State or Graph Based Test:

- Graph based testing also called as **state graph based testing**.
- State Graphs provide a framework for model based testing. After arriving at an executable state graph model, we execute or simulate that state graph model with event sequences as test cases.
- This is done before starting the actual implementation phase. This would support for testing the system implementation (program) against the system specification (State Graph) and also, support for automatic generation of test cases for the implementation.
- In order to design test cases we use following steps:
  1. Understand the system.
  2. Identify states, inputs and guards.
  3. Create a state graph model of the application.
  4. Verify whether the State Graph that we modeled is correct in all details.
  5. Generate sequences of test actions.

State Transition Testing:

- This testing is used where some aspect of the system can be described in what is called a 'finite state machine'. This simply means that the system can be in a (finite) number of different states, and the transitions from one state to another are determined by the rules of the 'machine'. This is the model on which the system and the tests are based.
- Any system where you get a different output for the same input, depending on what has happened before, is a finite state system.
- One of the advantages of the state transition technique is that the model can be as detailed or as abstract as you need it to be. Where a part of the system is more important (that is, requires more testing) a greater depth of detail can be modeled. Where the system is less important (requires less testing), the model can use a single state to signify what would otherwise be a series of different states.

Examples:

- Consider an example of an ATM of the bank to demonstrate the steps on creating corresponding State Graph.
- Customer uses a bank ATM to check balances of his/her bank accounts, deposit funds, withdraw cash and/or transfer funds (use cases). ATM Technician provides maintenance and repairs to the ATM.

- For example, if you request to withdraw Rs.1000 from a bank ATM, you may be given cash. Later you may make exactly the same request but it may refuse to give you the money because of your insufficient balance. This later refusal is because the state of your bank account has changed from having sufficient funds to cover the withdrawal to having insufficient funds. The transaction that caused your account to change its state was probably the earlier withdrawal.
- A state diagram can represent a model from the point of view of the system, the account or the customer.

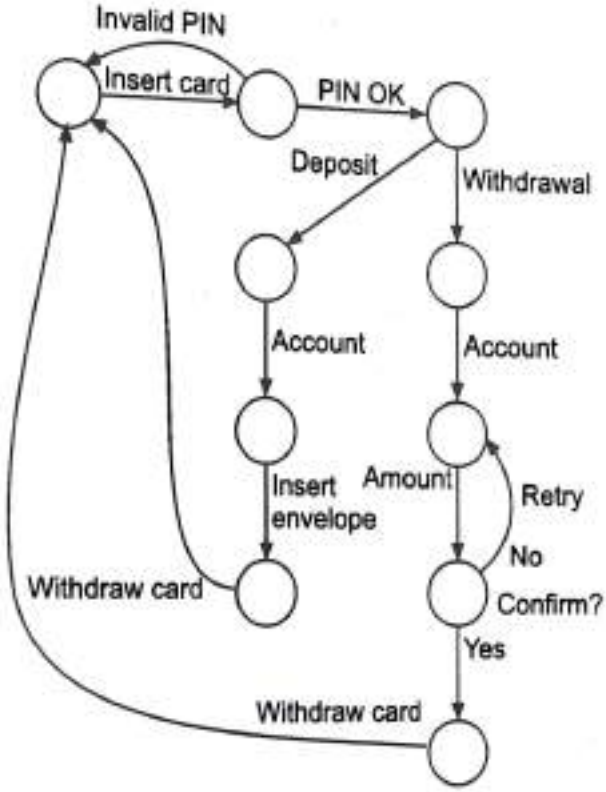


Fig. 2.10: State diagram for ATM system

- Let us consider another example of a word processor. If a document is open, you are able to close it. If no document is open, then 'Close' is not available. After you choose 'Close' once, you cannot choose it again for the same document unless you open that document. A document thus has two states: open and closed.

2.4.2 Advantages and Disadvantages

Advantages:

1. Black box testing is efficient when used on large systems.
2. The tester and developer in black box testing are independent of each other so testing is balanced and fair.
3. Testers can be non-technical.
4. There is no need for the tester to have detailed functional knowledge of the system.

5. Tests will be done from an end user's point of view, because the end user should accept the system.
6. Black box testing helps to identify vagueness and contradictions in functional specifications.
7. Test cases in black box testing can be designed as soon as the functional specifications are complete.

Disadvantages:

1. In black box testing, cases, it is challenging to design without having clear functional specifications.
2. It is difficult to identify tricky inputs if the test cases are not developed based on specifications.
3. It is difficult to identify all possible inputs in limited testing time. As a result, writing test cases may be slow and difficult.
4. There are chances of having unidentified paths during the testing process.
5. There is a high probability of repeating tests already performed by the programmer.

2.5 GRAY BOX TESTING [W-22; S-23]

- Gray Box Testing is a software testing method which is a combination of Black Box Testing method and White Box Testing method. In Black Box Testing, the internal structure of the item being tested is unknown to the tester and in White Box Testing the internal structure is known. In Gray Box Testing, the internal structure is partially known. This involves having access to internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black box level.
- Gray Box Testing is named so because the software program, in the eyes of the tester, is like a gray/semi-transparent box; inside which one can partially see.

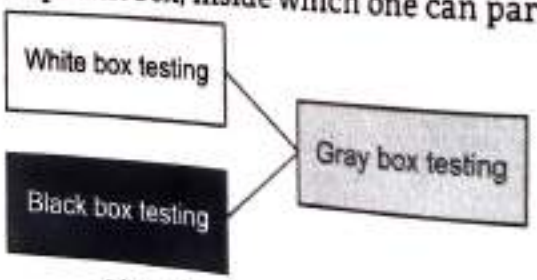


Fig 2.11: Gray Box Testing

- An example of Gray Box Testing would be when the codes for two units/modules are studied (White Box Testing method) for designing test cases and actual tests are conducted using the exposed interfaces (Black Box Testing method).

Levels Applicable to:

- Though Gray Box testing method may be used in other levels of testing, it is primarily useful in Integration Testing.

Need for Gray Box testing:

- Gray box testing is useful because it takes the straightforward technique of black box testing and combines it with the code-targeted systems in white box testing.
- Gray box testing is based on requirement test case generation because it presents all the conditions before the program is tested by using the assertion method.

2.5.1 Techniques of Gray Box Testing

- **Regression Testing:** This testing is performed after making a functional improvement or repair to the program. Its purpose is to determine if the change has regressed other aspects of the program. This can be accomplished by executing the following testing strategies.
  - **Retest all:** Rerun all existing test cases.
  - **Retest Risky Use Cases**: Choose baseline tests to rerun by risk.
  - **Retest by Profile**: Choose baseline tests to rerun by allocating time in proportion to operational profile.
  - **Retest Changed Segment:** Choose baseline tests to rerun by comparing code changes. (White box strategy).
  - **Retest within Firewall**: Choose baseline tests to rerun by analyzing dependencies. (White box strategy).
- **Pattern Testing:** It is best accomplished when historical data of previous system defects are analyzed. Your analysis will include specific reasons for the defect, which will require system analysis. Unlike black box testing where you are tracking the type of failures that are occurring, gray box testing digs within the code and determines why the failure happened. This information is extremely valuable, as future design of test cases will be proactive in finding the other failures before they hit production. Remember, having coding structure in place influences gray box test case design. Therefore, take advantage of powerful technique and design test cases.
- Analysis Template will include:
  1. The problem addressed.
  2. Applicable situation.
  3. The problem that can be discovered.
  4. Related problems.
  5. Solution for developers that can be implemented.
  6. Generic test cases.
- **New test cases will include:** 1. Security related test cases 2. Business related test cases 3. GUI related test cases 4. Language related test cases 5. Architecture related test cases 6. Database related test cases 7. Browser related test cases 8. Operating System related test cases.

- **Orthogonal Array Testing (OAT):** This testing is a statistical testing technique implemented by Taguchi. This method is extremely valuable for testing complex applications and e-commerce products. The e-commerce world presents interesting challenges for test case design and testing coverage. The black box testing technique will not adequately provide sufficient testing coverage. The underlying infrastructure connections between servers and legacy systems will not be understood by the black box testing team. A gray box testing team will have the necessary knowledge and combined with the power of statistical testing, an elaborate testing net can be set-up and implemented. Orthogonal Array Testing (OAT) can be used to reduce the number of combinations and provide maximum coverage with a minimum number of test cases. OAT is an array of values in which each column represents a variable - factor that can take a certain set of values called levels. Each row represents a test case. In OAT, the factors are combined pair-wise rather than representing all possible combinations of factors and levels.
- **Matrix testing:** It starts with developers defining all the variables that exist in their programs. Each variable will have an inherent technical risk, business risk and can be used with different frequency during it's' life cycle.

Advantages:

- Gray box testing offers combined benefits of both White box testing as well as Black box testing.
- Gray box testers rely on interface definition and functional specifications instead of source code.
- Gray box testers can design excellent test scenarios around communication protocols and data type handling due to limited information available.
- The testing will be performed from the user point of view instead of designer.
- Testing is done on the basis of high-level database diagrams and data flow diagrams.

Disadvantages:

- The ability to go over the code and test coverage is limited. Since the access to source code is not available.
- The tests can be redundant if the software designer has already run a test case.
- Testing every possible input stream is unrealistic because it would take an unreasonable amount of time; therefore, many program paths will go untested.
- Not suited for algorithm testing.

Uses:

- Gray box testing is a perfect fit for Web-based applications.
- Gray box testing is also the best approach for functional or domain testing.

2.5.2 Example of Gray Box Testing

- Let's take an example of web application testing. To explore gray testing will take a simple functionality of web application. You just want to enter email id as input in the web form and upon submitting the valid email id user and based on users interest (fields entered) user should get some articles over email. The validation of email is using JavaScript on client side only.
- In this case, if the tester doesn't know the internal structure of the implementations then you might test the web application of form with some cases like Valid Email ID, Invalid Email ID and based on this will check whether functionality is working or not.
- But tester is aware of some internal structure and if system is making the assumptions like:
  - System will not get Invalid email id.
  - System will not send email to invalid email id.
  - System will not receive failure email notifications.
- In this type of testing you have to test the application by disabling the JavaScript, it might be possible due to any reason JavaScript is failed and System get Invalid email to process and all assumptions made by system will failed, as a result incorrect inputs are send to system, so
  - System will get Invalid email id to process.
  - System will send email to invalid email id.
  - System will receive failure email notifications.

2.5.3 Comparison between three forms of Testing Techniques

Table 2.4: Comparison of Testing Techniques

Sr. No.	Black Box Testing	Gray Box Testing	White Box Testing
1.	The Internal Workings of an application are not required to be known	Some knowledge of the internal workings is known.	Tester has full knowledge of the Internal workings of the application.
2.	Also known as closed box testing, data driven testing and functional testing.	Another term for Gray box testing is translucent testing as the tester has limited knowledge of the insides of the application.	Also known as clear box testing, structural testing or code based testing.

Contd...

3.	Performed by end users and also by testers and developers.	Performed by end users and also by testers and developers.	Normally done by testers and developers.
4.	Testing is based on external expectations. Internal behaviour of the application is unknown.	Testing is done on the basis of high level database diagrams and data flow diagrams.	Internal workings are fully known and the tester can design test data accordingly.
5.	This is the least time consuming and exhaustive.	Partly time consuming and exhaustive.	The most exhaustive and time consuming type of testing.
6.	This can only be done by trial and error method.	Data domains and Internal boundaries can be tested, if known.	Data domains and Internal boundaries can be better tested.

Summary

- A system is tested for defined structure and required functionalities.
- Structural testing is called as white box testing and functional testing is called as black box testing.
- White box and Black box testing are applied once the code is ready.
- Test cases are designed to validate the working module against the expected structure and functionalities.
- In this chapter, we have covered various white box and black box testing strategies with relevant example(s), gray box testing and applicability.

Check Your Understanding

1. Which of the following is NOT a test case design technique?
   (a) Black box (b) White box
   (c) Cyclomatic Complexity (d) Top-down
2. Test cases are nothing but_____.
   (a) set of input values (b) stubs
   (c) driver (d) boundaries
3. White box testing is called as _____.
   (a) statement testing (b) structural testing
   (c) functional testing (d) behavioral testing
4. Which of the following is a black box testing?
   (a) Equivalence partitioning (b) Boundary value analysis
   (c) Decision testing (d) All of the above

5. Which testing method says that "Every statement in the component is executed at least once"?
   (a) Black box testing
   (b) White box testing
   (c) Gray box testing
   (d) All of the above
6. Which of the following is the formula for Conditional Coverage?
   (a) (Number of decision outcomes tested/Total number of decision outcomes) × 100%.
   (b) (Total decision exercised/Total number of decisions) × 100%.
   (c) (Number of statements exercised/Total number of statements) × 100%.
   (d) (Total path exercised/Total number of paths in program) × 100%.
7. _____ is the testing type for checking the modified code after fixing the bug/defect.
   (a) Mutation testing
   (b) Integration testing
   (c) Gray box testing
   (d) Top-down
8. Which test case design technique uses boundaries for input values?
   (a) Boundary value analysis
   (b) Equivalent Partitioning
   (c) Decision table
   (d) Regression testing
9. _____ technique divides the input domain of a program into classes of data. These classes or sets are used to derive the test cases.
   (a) Equivalent Partitioning
   (b) Boundary value analysis
   (c) Cyclomatic Complexity
   (d) Coverage testing
10. Cyclomatic complexity is used for _____.
    (a) knowing the logical complexity.
    (b) to know the number of paths to be executed.
    (c) to ensure that all the nodes are tested.
    (d) All of the above.

ANSWERS

1. (d)	2. (a)	3. (b)	4. (d)	5. (b)	6. (b)	7. (a)	8. (a)	9. (a)	10. (d)

Practice Questions

Q.I Answer the following questions in short.

1. What is Cyclomatic Complexity? Explain with example.
2. Write a brief note on Gray box testing
3. What is Boundary Value Analysis?
4. What is structural testing?
5. What is functional testing?

Q.II Answer the following questions.

1. What is a White Box testing? Explain with its techniques.
2. Explain the Black Box testing with any two techniques.
3. Write advantages and disadvantages of following tests:
   White Box testing, Black box testing, Gray Box testing.
4. Write difference between White box test and Black box test.
5. Explain the term Unit Testing.
6. Write in detail about various test case design techniques.
7. Describe an Equivalence Partitioning?
8. Explain the coverage testing.
9. Write techniques of Gray Box testing?
10. Write difference between Static and Dynamic testing.

Q.III Define the terms.

1. State transition
2. Decision table
3. Cause Effect
4. Behavioral testing
5. Execution testing
6. Mutation testing

★★★

3...

Software Testing Strategies and Software Metrics

Learning Objectives ...

- To learn about Software Testing process.
- To get knowledge of Unit Testing, Integration Testing, Acceptance Testing, System Testing.
- To learn about various approaches of testing.
- To know about Regression, Performance, Smoke and Load testing.

3.1 INTRODUCTION

- This chapter describes several approaches to testing software. Software testing must be planned carefully to avoid unexpected loss in terms of cost, time and resources.
- The choice of test approaches or test strategy is one of the most powerful factors in the success of the test effort and the accuracy of the test plans and estimates. This factor is under the control of the testers and test leaders.

3.2 SOFTWARE TESTING PROCESS

- Like the software development process (where a software/application undergoes various development phases), the system also has to move through different levels of testing before releasing it to the user. Testing process starts immediately once the requirements are gathered for the development of software. In the beginning, the system's gathered requirements are verified against the specifications. Once the system's low-level and high-level designing is ready then it is converted into coding from where the validation of the system under test starts.
- Every system has to be tested in order to verify and validate the system's structure as well as system's functionality. Entire testing activity is associated with what to test,

how to test and in what way? What to test is referred as whether to test the structure of code or functionality of code i.e. the tester has to perform structural testing or functional testing. Testing is to be done in a static and dynamic way.

3.2.1 Testing Strategies

- Fully developed and functional software is made up of several small functional modules. Each module is designed to perform certain defined functionality. It is obvious that if you test a complete functional software system at its final stage instead of testing at intermediate level then it will increase the cost considerably as you may find lots of error at the stage of software release. At this stage, it could be very difficult to locate that error along with the module which is causing such error(s). Hence debugging would be difficult. Testing levels are implemented to correct the code at the module level, at boundary level and then as a complete system level.
- Once the code is ready then it passes through these different testing levels. These levels are called as **testing strategies**.
- Various testing strategies are:
  1. Unit or Component level.
  2. Integration testing at small.
  3. Validation Testing.
  4. System Testing.
  5. Integration Testing at Large.
  6. Acceptance Testing.

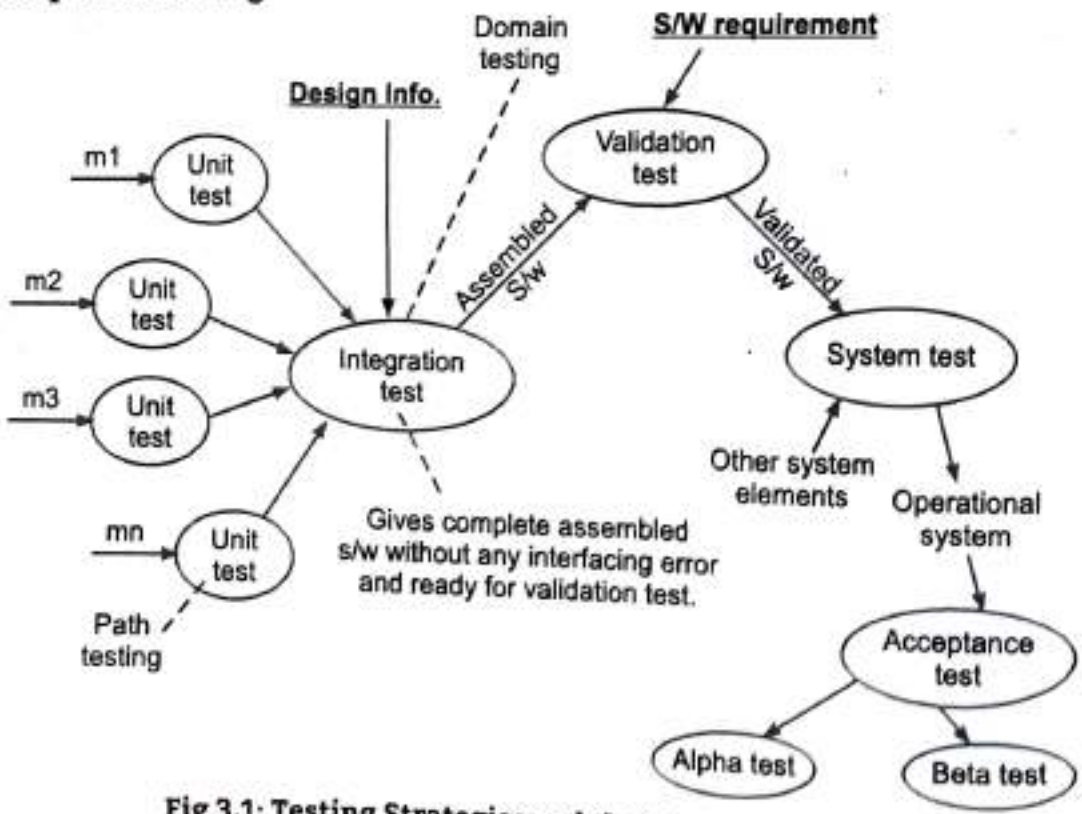


Fig 3.1: Testing Strategies and their Interconnection

- The basic levels are Unit testing, Integration testing, System testing and Acceptance testing. These different levels of testing attempt to detect different types of faults. Every level of testing is verified against certain expected value(s). Acceptance testing is the final level of testing before its release and so tested against the true customer needs. System level testing checks the system against the defined system requirements. Integration testing ensures the integration of tested units (small modules) as per the low and high level design standards.

3.3 UNIT TESTING

[S-22]

Levels of Software Testing:

- We already know that software testing is a process of executing a program or application with the intent of finding the software bugs. Software testing levels include the different methodologies that can be used while conducting software testing. Different levels of testing are used in the testing process, each level of testing aims to test different features of the software system.
- The main levels of software testing are Functional testing, which refers to activities that verify a specific action or function of the code. Functional tests tend to answer the question of "can the user do this" or "does this particular feature work" and Non-functional testing refers to aspects of the software that may not be related to a specific function or user action, such as scalability or other performance, behavior under certain constraints, or security.
- Execution-based software testing especially for large systems are usually carried out at different levels. In most cases, there will be 3 to 4 levels or major phases of testing i.e. unit test, integration test, system test, acceptance test as shown in Fig. 3.2.

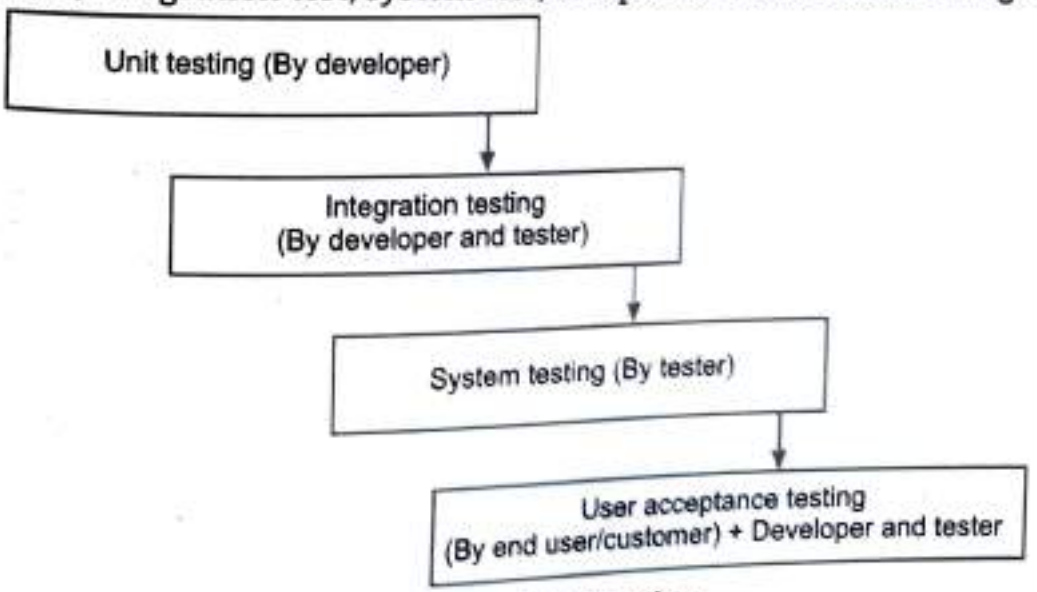


Fig. 3.2: Levels of Testing

- Following are the main levels of software testing:

1. **Unit Testing** is a level of the software testing process where individual units/components of a software/system are tested. The purpose is to validate that each unit of the software performs as designed.

2. **Integration Testing** is a level of the software testing process where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.
3. **System Testing** is a level of the software testing process where a complete, integrated system/software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.
4. **Acceptance Testing** is a level of the software testing process where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

- Testing that occurs at the lowest level is called **Unit testing** or Module testing. Unit testing is performed to test the individual units of software. Unit is the smallest part of software system which is testable. It may include code files, classes, and methods which can be tested individually for correctness.
- The primary goal of unit testing is to take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as you expect. In this testing, each unit is tested separately before integrating them into modules to test the interfaces between modules.

Functions:

- Unit testing performs the following functions:
  1. Unit testing tests all control paths to uncover maximum errors that occur during the execution of conditions present in the unit being tested.
  2. Unit testing confirms that all statements in the unit have been executed at least once.
  3. It tests data structures that represent relationships among individual data elements.
  4. Unit testing checks the range of inputs given to units i.e. a maximum and minimum value.

Concept of Driver and Stub: **[W-22; S-23]**

- The most common approach to unit testing requires drivers and stubs to be written. Drivers and stubs are special-purpose arrangements, generally code, required to test units individually which can act as an input to the unit/module and can take output from unit/module.
- The driver simulates a *calling unit* and the stub simulates a *called unit*. A component is not a stand-alone program, driver and/or stub software must often be developed for each unit test.

- In most applications, a driver is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results.
- Stubs serve to replace modules that are subordinate (invoked by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.
- Both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product.
- Fig. 3.3 shows Stub and Driver concept in Unit testing.

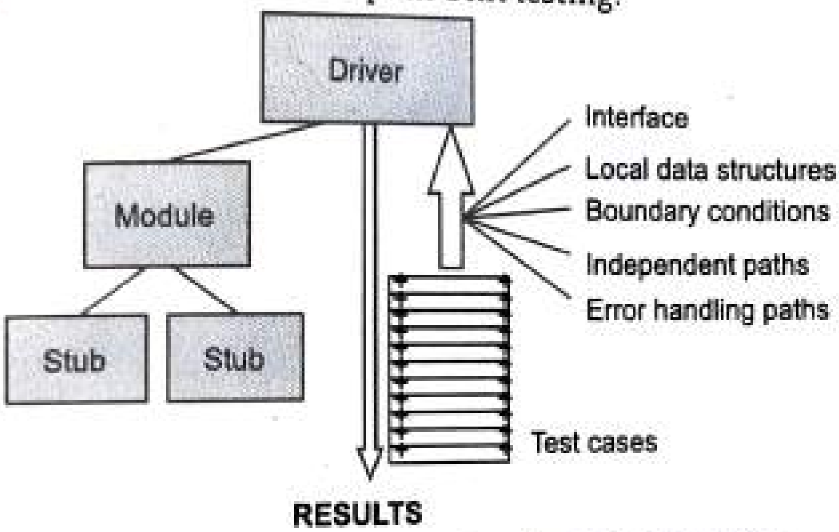


Fig. 3.3: Concept of Stub and Driver in Unit Testing

Examples:

- **A test driver can replace the real software and more efficiently test a low-level module:** Drivers send test-case data to the modules under test, read back the results, and verify that they're correct.
- **A test stub sends test data up to the module being tested:** A low-level interface module is used to collect temperature data from an electronic thermometer. A display module sits right above the interface, reads the data from the interface, and displays it to the user. To test the top-level display module, you'd need blow torches, water, ice, and a deep freeze to change the temperature of the sensor and have that data passed up the line.

Table 3.1: Difference between Driver and Stub

Sr. No.	Driver	Stub
1.	Drivers mainly created for integration testing such as the bottom-up approach.	Stubs are mainly created for integration testing such as the top-down approach.

Contd...

2.	A driver is basically a program that accepts test case data and passes that data to the module that is being tested.	Stubs are programs that are used to replace modules that are subordinate to the module to be tested.
3.	A driver is a piece of software which calls the functions in the unit under test.	A stub is a small program routine that substitutes for a longer program, possibly to be loaded later or that is located remotely.
4.	Driver is a piece of code emulating a calling function.	Stub is the piece of code emulating the called function.

Advantages of Unit Testing:

1. Unit testing provides a sort of living documentation of the system. Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain a basic understanding of the unit's interface (API).
2. Unit testing may reduce uncertainty in the units themselves and can be used in a bottom-up testing style approach.
3. Unit tests find problems early in the development cycle.
4. Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly.

Disadvantages of Unit Testing:

1. The biggest disadvantage of unit testing is the initial time required to develop them.
2. Testing will not catch every error in the program, since it cannot evaluate every execution path in any but the most trivial programs. The same is true for unit testing.
3. Unit testing by definition only tests the functionality of the units themselves. Therefore, it will not catch integration errors or broader system-level errors.
4. Unit testing should be done in conjunction with other software testing activities, as they can only show the presence or absence of particular errors; they cannot prove a complete absence of errors.
5. Another problem related to writing the unit tests is the difficulty of setting up realistic and useful tests.

3.4 INTEGRATION TESTING

- Integration is a process by which components are aggregated to create larger components.
- Testing the data flow or interface between two features is known as **Integration testing.**

- Testing that occurs at the lowest level is called unit/module testing. As the units are tested and the low-level bugs are found and fixed, they are integrated and integration testing is performed against groups of modules. Integration testing may start at module level where different units and components come together to form a module and so up to system level.
- Integration testing is considered as structural testing. Fig. 3.4 shows a system schematically. It mainly focuses on I/O protocols, parameters passing between different units, modules and/or system etc.
- The process of incremental testing continues putting together more and more pieces of the software until the entire product or at least a major portion of it is tested at once in a process, called as System testing. With this testing strategy, it's much easier to isolate bugs. When a problem is found at the unit level, the problem must be in that unit. If a bug is found when multiple units are integrated, it must be related to how the modules interact. There are exceptions to this but by the large testing and debugging step by step is much more efficient than testing everything at once.

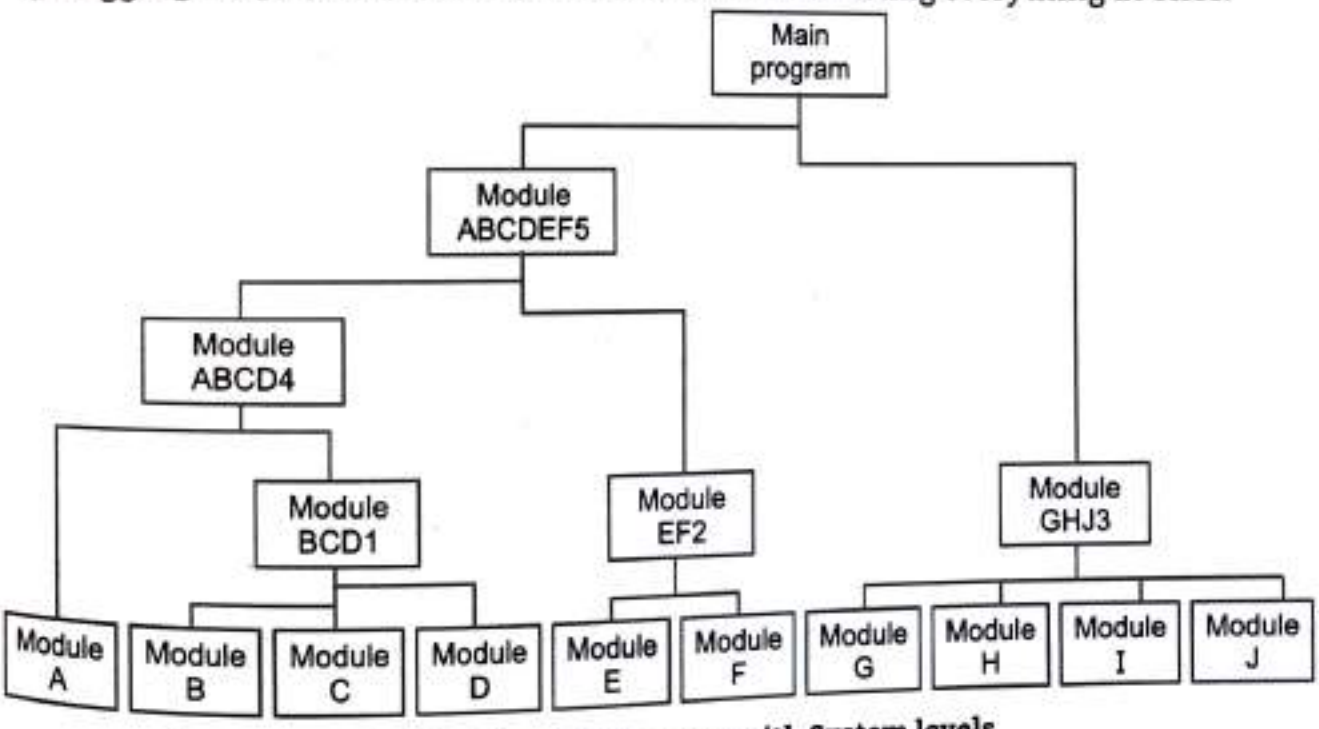


Fig. 3.4: Integration testing with System levels

Working of Integration Testing:

- Once, unit testing is complete, integration testing begins. In integration testing, the units validated during unit testing are combined to form a subsystem.
- The integration testing is aimed at ensuring that all the modules work properly as per the user requirements when they are put together i.e. integrated.
- The objective of integration testing is to take all the tested individual modules, integrate them, test them again, and develop the software, which is according to design specifications.

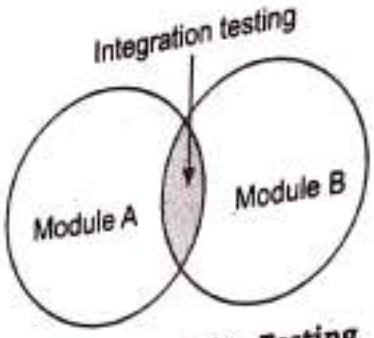


Fig. 3.5: Integration Testing

Advantages of Integration Testing:

1. The unit modules may not be tested as per the changes made to the requirements which will lead to error page when Integration Testing is done.
2. It is easy to fix the error in the Integration Testing when compared to the System testing.
3. The interfaces should be checked thoroughly if any error messages are shown.

Decomposition-based Integration Testing:

- The basic idea for this type of integration is based on the decomposition of design into functional components or modules.
- In the tree designed for decomposition-based integration, the nodes present the modules present in the system and the links/edges between the two modules represent the calling sequence. The nodes on the last level in the tree are leaf nodes.
- Thus, with the decomposition-based integration, we want to test the interfaces among separately tested modules. Integration methods in decomposition-based integration depend on the methods on which the activity of integration is based.
- The two methods are:
  1. Method of integrating is to integrate all the modules together and then test it.
  2. Method is to integrate the modules one by one and test them incrementally.
- Based on above methods, integration testing methods are classified into two categories i.e. Non-incremental and Incremental.

Advantages of Decomposition-based Integration:

1. Better for monitoring the decomposition tree.
2. Debugging is easy and simple in decomposition-based integration.

Disadvantages of Decomposition-based Integration:

1. More efforts required for this testing.
2. Stubs and drivers are needed for testing.
3. Drivers are more complicated to design.

Types of Integration Testing:

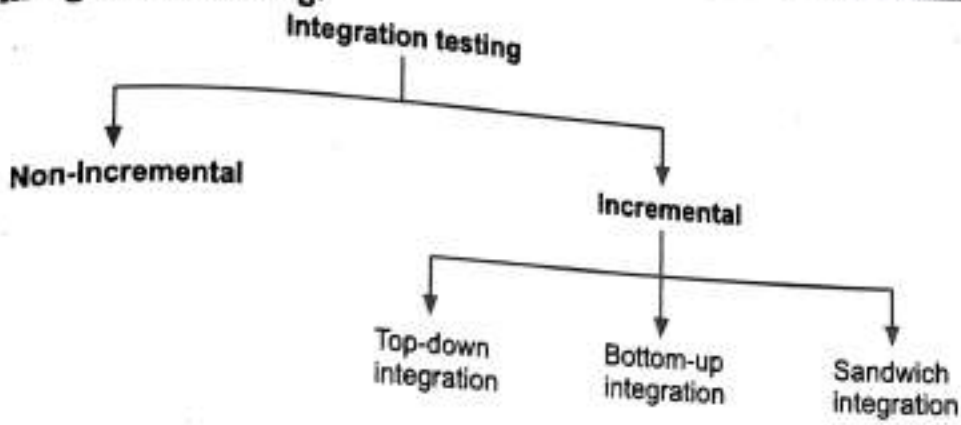


Fig. 3.6: Types of Integration Testing

3.4.1 Top-down Approach

[S-22]

- It is an incremental approach to construction of program structure. Modules/Components are integrated by moving downward through control hierarchy, beginning with the main control module (main program).
- Modules/Components subordinate to main control module are integrated in either Depth-first manner or Breadth-first manner.

Example 1: Based on Depth-first Integration:

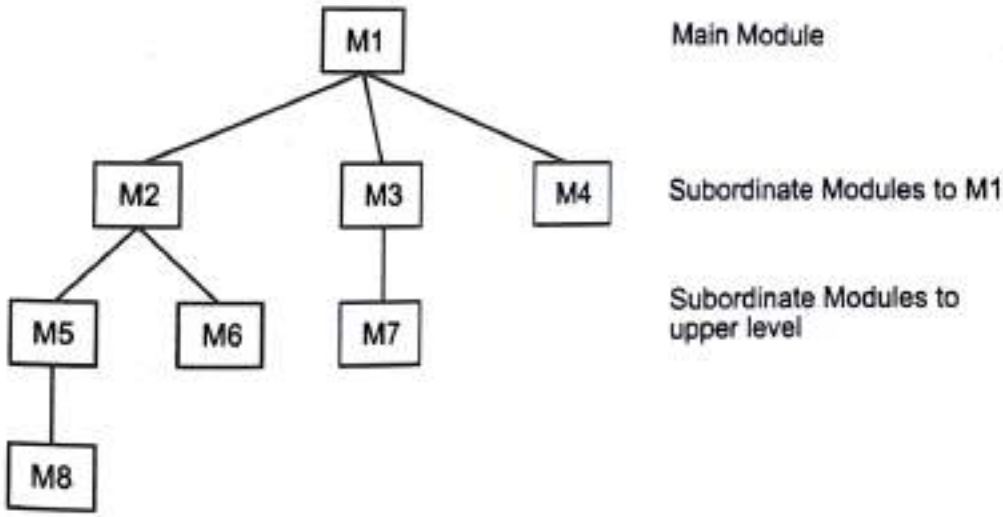


Fig. 3.7 (a): Top-Down Integration (Depth-First)

- Referring to Fig. 3.7(a), Depth-first integration would integrate all components on control path of the structure. For example, selecting the left-hand path, components M1, M2, M5 would be integrated first. Next M8 or M6 would be integrated. Then the central and right-hand control paths are built.
- "With Depth-first Integration, a complete function of the software is implemented and demonstrated".

- On the other hand, Breadth-first integration integrates all components directly subordinate at each level. For example, components M1, M2, M3 and M4 would be integrated first. At the next control level (M5, M6) would be integrated, and so on.

Example 2: Based on Breadth-first Integration:

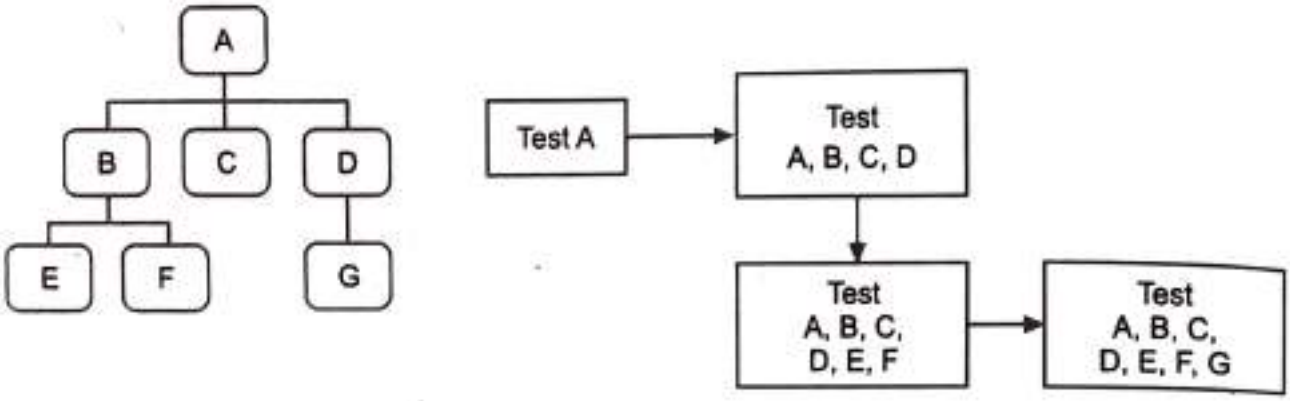


Fig. 3.7 (b): Top-Down Integration (Breadth-First)

Process of Top-down Integration:

Step 1: The main control module is used as a test driver, and stubs are substituted for all components directly subordinate to the main control module.

Step 2: Depending on the integration approach selected, subordinate stubs are replaced one at a time with actual components.

Step 3: Tests are conducted as each component is integrated.

Step 4: On completion of each set of tests, another stub is replaced with a real component.

Step 5: Retesting may be conducted to ensure that new errors have not been introduced.

- This process continues from step 2 until the entire program structure is built.

Advantages:

1. Any design faults or major questions about functional feasibility can be addressed at the beginning of testing instead of the end.

Disadvantages:

1. Writing stubs can be difficult. Stubs must allow all possible conditions to be tested.
2. Large number of stubs may be required, particularly if the lowest level of the system contains many methods.
3. Stubs replace lower-level modules at the beginning of top-down process; so no significant data can flow upward. In this situation, tester is left with 3 choices:
   - Delay many tests until stubs are replaced with actual modules.
   - Develop stubs that perform limited functions that simulate the actual module.

- Integrate the software from the top of hierarchy to bottomward (Top-down Integration) but will depend on stub modules.

3.4.2 Bottom-up Approach

[S-22]

- Bottom-up integration is just the opposite of top-down integration. In this approach, the components for a new product development become available in reverse order, starting from bottom. That means this approach begins construction and testing with components at the lowest levels in program structure.
- As the components are integrated from the bottom, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

Process of Bottom-up Integration:

Step 1: Low-level components are tested individually by writing a driver to coordinate test case input and output.

Step 2: Drivers are removed and tested components are integrated, moving upward in program structure.

Step 3: Tests are conducted as each component is integrated.

Step 4: This is done repeatedly until the entire program structure is built.

Example 3: Based on Bottom-up integration:

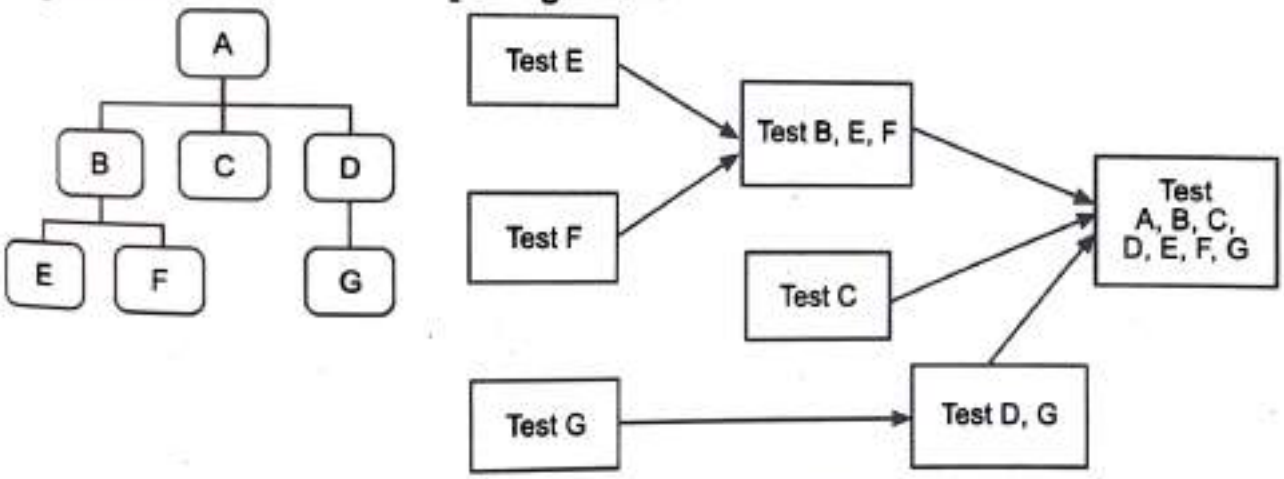


Fig. 3.8: Bottom-Up Integration

Disadvantages:

1. The program as an entity does not exist until the last module is added.
2. Bad for functionally decomposed systems as it tests the main module last.
3. Interface errors are discovered late.

3.5 SYSTEM TESTING

- Once the entire system has been built then it has to be tested against the "System Specifications". System testing checks if it delivers the required features. System testing verifies the entire product, after integrating all software and hardware components and validates it according to original project requirements.

- A testing group performs system testing before the product is made available to the customer. It also verifies the software operation from the viewpoint of the end user with different configurations or set-up.
- It can begin whenever the product has sufficient functionality to execute some of the tests or after unit and integration testing are completed.
- It is a series of different tests whose primary purpose is to fully workout the computer-based system.
- System testing comprises two types as illustrated in Fig. 3.9.

3.5.1 Types of System Testing

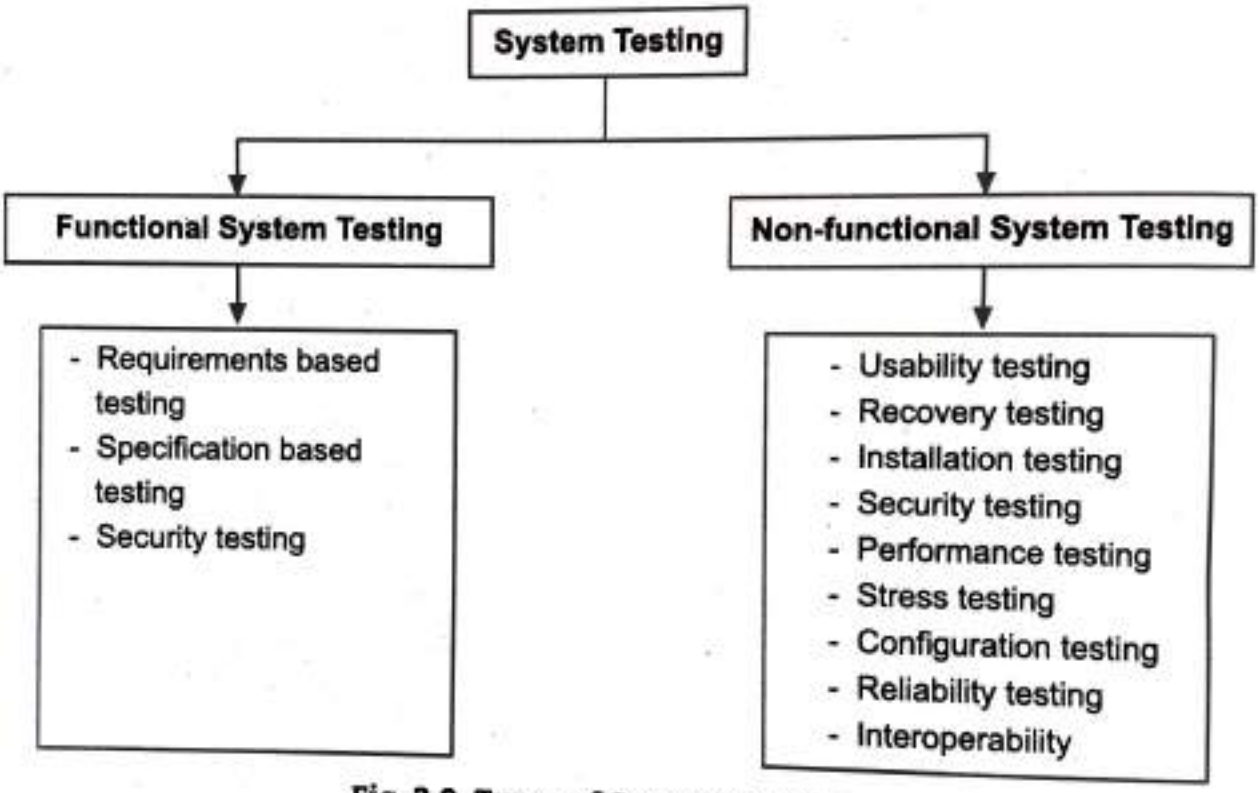


Fig. 3.9: Types of System Testing

(a) Functional System Testing:

- Functional testing involves testing a product's functionality and features. It is conducted for functional requirements.
  - Functional requirements affect the functionality of the system.
  - Functional requirements describe what the system should do?
- In other words, it is the process of validating an application or Web site to verify that each feature complies with its specifications and correctly performs all its required functions. This involves a series of tests, which tests the functionality by using a wide range of normal and erroneous input data. This can involve testing of the product's user interface, APIs, database management, security etc. Testing can be performed on an automated or manual basis using black box methodologies.

Types of Functional System Testing:

1. **Requirements-based testing:** This testing technique validates that the requirements are correct, complete, clear, and logically consistent. This allows designing a necessary and sufficient set of test cases from those requirements. This testing process is performed by QA teams.
2. **Specification-based testing:** Specification-based testing uses the specification of the program as the point of reference for test data selection and adequacy. Specification means any one of these things: (Written document/Collection of user scenarios (use-cases)/Set of models/Formal, mathematical description/Prototype).
3. **Security testing:** A process to determine that an information system protects data and maintains functionality as intended. It can be performed by testing teams or by specialized security-testing professionals.

(b) Non-functional System Testing:

- Non-functional testing involves testing the product's quality factors such as reliability, scalability etc. It is conducted for non-functional requirements. These requirements are global constraints on a software system.
- Non-functional testing requires the expected results to be documented in qualitative and quantifiable terms. It requires a large amount of resources and the results are different for different configurations and resources.

Types of Non-functional System Testing:

1. **Usability Testing**: This testing technique verifies the comfort with which a user can learn to operate, prepare inputs for, and take outputs of a system or component. This testing process is usually performed by end users.
2. **Recovery Testing**: This testing technique evaluates how well a system recovers from crashes, hardware failures, or other catastrophic problems. This testing process is performed by the testing team.
3. **Installation Testing:** Quality assurance work that focuses on what customers will need to do to install and set up the new software successfully. It may involve full, partial or upgrades install/uninstall processes and are typically done by a software testing engineer in conjunction with the configuration manager.
4. **Security Testing:** (As mentioned in the 'Types of Functional System Testing' section)
5. **Stress Testing:** A testing technique which evaluates a system or component at or beyond the limits of its specified requirements. This testing process is usually conducted by the performance engineer.
6. **Performance Testing:** Functional testing conducted to evaluate the compliance of a system or component with specified performance requirements. This testing process is usually conducted by the performance engineer.

7. **Configuration Testing:** This testing technique determines minimal and optimal configuration of hardware and software, and the effect of adding or modifying resources such as memory, disk drives and CPU. This testing process is performed by the performance test engineers.

3.6 ACCEPTANCE TESTING

[S-22, 23; W-22]

- Acceptance Testing is often the final step before rolling out the application. It is performed to determine whether system meets user requirements or not.
- Usually the end users, who will be using the application, test the application before 'accepting' it. This type of testing gives the end user confidence that the application being delivered to them meets their requirements.
- Acceptance tests can range from an informal "test drive" to a planned and systematically executed series of tests.
- With respect to the website or application produced, the client or end user "reviews" and "tests" the application for any non-conformity to requirement specifications or any update which are not included in original specifications.
- Customer may write the acceptance test criteria and request the organization to execute them, or the organization may produce an acceptance testing criteria, which is to be approved by the customer.

3.6.1 Acceptance Criteria

- There are three Acceptance criteria:

1. Acceptance Criteria – Product Acceptance:

- During the requirements phase, each requirement is associated with acceptance criteria. Whenever there are changes to requirements, the acceptance criteria are accordingly modified and maintained. While accepting the product, end user executes all existing test cases to determine whether they meet the requirements.

2. Acceptance Criteria – Procedure Acceptance:

- Acceptance criteria can be defined based on the procedures followed for delivery. Some of the examples of such acceptance criteria are:
  - User manuals, administration and troubleshooting documentation should be part of the release.
  - Along with the executable file(s), source code of the product should be delivered in a CD.
  - A minimum of 10 to 15 employees should be trained on the product usage prior to deployment.

3. **Acceptance Criteria – Service Level Agreements:**

- Service level agreements can become part of acceptance criteria. A Service Level Agreement is a document or contract signed by two parties: the product organization and the customer.
- Example could be:
  - All major defects that come-up during first six months of deployment need to be fixed free of cost.
  - All major defects are to be fixed within 48 hours of reporting.

3.6.2 Types of Acceptance Testing [S-22]

- There are two kinds of acceptance testing: Alpha and Beta Testing.
- It is virtually impossible for a software developer to see how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data might be regularly used; output that seemed clear to the tester may be unintelligible to a user in the field.
- When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements.
- Conducted by the end user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executable series of tests.
- In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

1. **Alpha Testing:**

- Alpha testing usually comes after system testing and involves both white and black box testing techniques. Company employees test the software for functionality and give feedback. After this testing phase, any functions and features may be added to the software, which needs to be tested again in a conventional manner.
- Main features of Alpha testing are:
  1. Outside users are not involved while testing.
  2. White box and black box practices are used.
  3. Developers are involved.
- Alpha testing is performed at the developer's site, with the presence of developer checking over the customer's shoulder as they use the system to determine errors. This testing is considered as a form of internal acceptance testing in which user tests the software at developer's site. In other words, this testing assesses the performance of the software in the environment in which it is developed.

- On completion of alpha testing, users report errors to software developers so th they can correct them.
- Fig. 3.10 shows the Approach of Alpha testing.

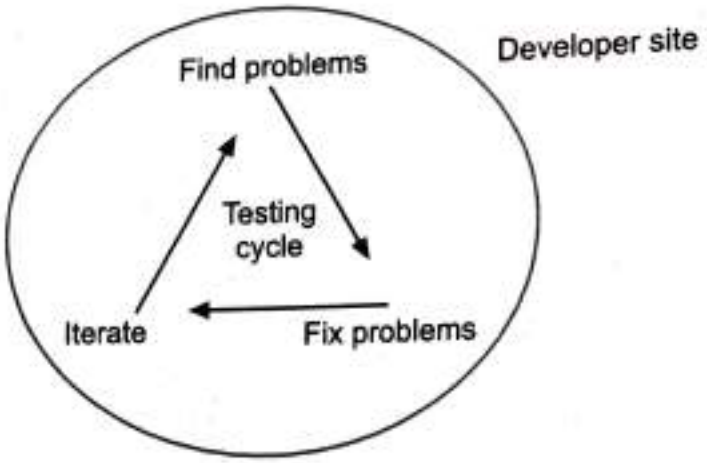


Fig. 3.10: Alpha testing Approach

Advantages of Alpha Testing:

1. It provides a better view about the reliability of the software before delivering it t the customer.
2. It simulates real time user behavior and environment.
3. It detects many serious errors.
4. It has the ability to provide early detection of errors with respect to design an functionality.

Disadvantages of Alpha Testing:

1. Change in customer needs may extend the testing success.
2. Sometimes, developers and testers are dissatisfied with the results of alph testing.

2. Beta Testing:

- Beta testing is the term used to describe the external testing process in whic software is sent out to a select group of potential customers who use it in a real-worl environment. This testing usually occurs towards the end of the product developmen cycle and ideally should just be a validation that the software is ready to release t real customers.
- Beta tests can be a good way to find compatibility and configuration bugs.
- Usability testing is another area that beta testing can contribute to if the participant is well chosen a good mix of experienced and inexperienced users. Those users wh will be using the software for the first time can readily find anything that's confusin or difficult to use.

- Beta testing is 'live' testing and is conducted in an environment, which is not controlled by the developer. That is, this testing is performed without any interference from the developer.
- Beta testing is performed to know whether the developed software satisfies user requirements and fits within the business processes.
- Both alpha and beta testing are very important while checking the software functionality and are necessary to make sure that all users' requirements are met in the most efficient way.

Advantages:

1. Beta testing allows a company to test post-launch infrastructure.
2. Reduces product failure risk via customer validation.
3. Improves product quality via customer feedback.
4. Cost effective compared to similar data gathering methods.
5. Creates goodwill with customers and increases customer satisfaction.

Disadvantages:

1. Test management is an issue. As compared to other testing types which are usually executed inside a company in a controlled environment, beta testing is executed out in the real world where you rarely have any control.
2. Finding the right beta users and maintaining their participation could be a challenge.

Alpha Testing versus Beta Testing:

Table 3.2: Comparison of Alpha and Beta testing

Sr. No.	Alpha Testing	Beta Testing
1.	Alpha testing performed by testers who are usually internal employees of the organization.	Beta testing is performed by Clients or end users who are not employees of the organization.
2.	Alpha testing is performed at the developer's site.	Beta testing is performed at client location or end user of the product.
3.	Reliability and security testing are not performed in alpha testing.	Reliability, security, robustness are checked during beta testing.
4.	Alpha testing involves both the white box and black box techniques.	Beta testing typically uses black box testing.

Contd...

5.	Alpha testing requires a lab environment or testing environment.	environment. Software is made available to the public and is tested in a real time environment.
6.	Long execution cycle may be required for alpha testing depending upon the software.	Only a few weeks of execution are required for beta testing depending upon the software.
7.	Critical issues or fixes can be addressed by developers immediately in alpha testing.	Most of the issues or feedback collected during Beta testing is forwarded to developers which are taken care before final release of the product.
8.	Alpha testing is to ensure the quality of the product before moving to beta testing.	Beta testing also concentrates on quality of the product, but gathers user input on the product and ensures that the product is ready for real time users.

3.7 VALIDATION AND VERIFICATION

3.7.1 Validation

[W-22

- Validation testing is the process of evaluating a system or component during or end of the development process to determine whether it satisfies specified requirements or not.

Validation Activities:

- Validation activities can be divided into the following:
  1. **Low level testing:**
     (i) Unit (Module) testing.
     (ii) Integration testing.
  2. **High level testing:**
     (i) System testing.
     (ii) Acceptance testing.
     (iii) Scenario testing.
     (iv) Regression testing.
     (v) Ad hoc testing.
     (vi) Automation testing.

Validation Methods:

- Validation methods are decomposed into following methods:
  - **Black box methods for function-based tests :**
    (i) Equivalence partitioning.
    (ii) Boundary value analysis.
    (iii) State transition testing.
  - **White box methods for structure-based tests :**
    (i) Statement coverage.
    (ii) Decision (Branch) coverage.
    (iii) Path coverage.
    (iv) Function coverage.
    (v) Mutation testing.

Levels of Validation:

- There are four levels of validation:

(i) **Component Testing:**

- Testing conducted to verify the implementation of the design for one software element (unit, module) or a collection of software elements.

(ii) **Integration Testing:**

- An orderly progression of testing in which various software elements and/or hardware elements are integrated together and tested. This testing proceeds until the entire system has been integrated.

(iii) **System Testing:**

- The process of testing an integrated hardware and software system is to verify that the system meets its specified requirements.

(iv) **Acceptance Testing:**

- Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system.

3.7.2 Verification [S-22]

- Verification is the process of evaluating work-products of a development phase to determine whether they meet the specified requirements. This process ensures that the product is built according to the requirements and design specifications. It also answers to the question, 'Are we building the product right?'

Verification Testing - Workflow:

- Verification testing can be best demonstrated by using V-Model. The artifacts such as Test Plans, Requirement Specification, Design, Code and Test Cases are evaluated.

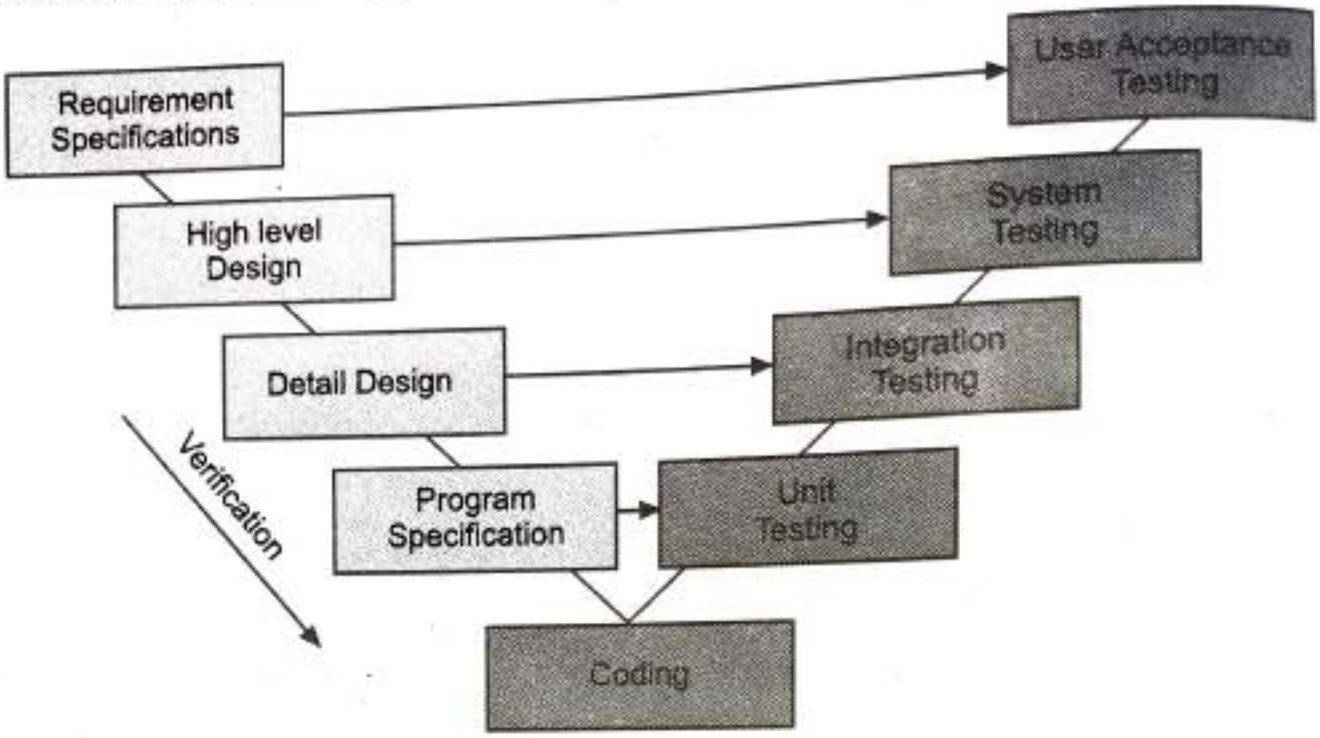


Fig. 3.11: V-model of VERIFICATION

Types of Verification:

- There are four types of verification that can be applied to the various levels:
  - (i) **Inspection:** Typical techniques include desk checking, walkthroughs, software reviews, technical reviews, and formal inspections (e.g., Fagan approach).
  - (ii) **Analysis:** Mathematical verification of the test item, which can include estimation of execution times and estimation of system resources.
  - (iii) **Code review:** Also known as "logic-driven" testing. Given input values are traced through the test item to assure that they generate the expected output values with the expected intermediate values along the way. Typical techniques include statement coverage, condition coverage, and decision coverage.
  - (iv) **Prototype:** Also known as "input/output driven" testing. Given input values are entered and the resulting output values are compared against the expected output values.

3.8 BIG BANG APPROACH (NON-INCREMENTAL TESTING) [W-22; S-23]

- In this type, all components are combined at once to form a program. That is, test all components in isolation and then mix them all together.

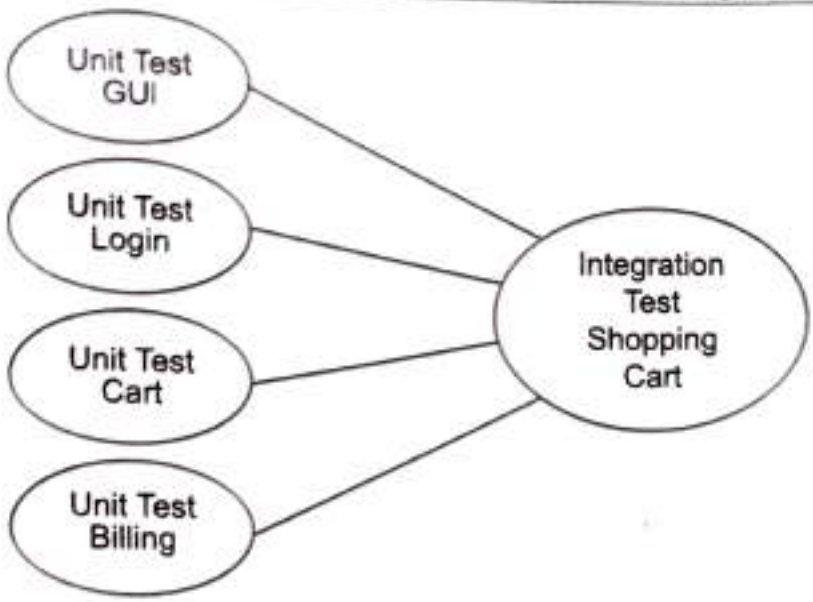


Fig. 3.12: Big Bang Approach

- The non-incremental approach is also known as "Big-Bang" testing.
- This approach is great with smaller systems, but it can end up taking a lot of time for larger, more complex systems. However, this approach is not recommended as both drivers and stubs are required.
- Biggest disadvantage of this approach is that Debugging is not easy. It is difficult to trace the cause of any failure since all components are merged all at once, which results complexity in correction.
- A second disadvantage is that you cannot start integration testing until all the modules have been successfully unit tested. Some modules that work together may be completed early, but they cannot be integration tested until all the modules in the application are completed.

3.9 SANDWICH APPROACH [W-22; S-23]

- This approach is also called as "Bi-directional integration testing". Sandwich testing defines testing into two parts and follows both parts starting from both ends, i.e., top-down approach and bottom-up approach either simultaneously or one after another. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready.
- Sandwich approach overcomes this shortcoming of top-down and bottom-up approaches. In the sandwich integration approach, testing can start as and when modules become available. Therefore, this is one of the most commonly used integration testing approach.

- Sandwich integration testing is also a vertical incremental [illegible] that tests the bottom and top layers and tests the integrated system in the computer software development process. Using stubs, it tests the user interface in isolation as well as tests the lower level functions using drivers.
- The system is viewed as having three layers:
  1. A target layer in the middle.
  2. A layer above the target.
  3. A layer below the target.
- Testing converges at the target layer. Top-down approach is used for the upper layers. Bottom-up approach is used for the subordinate layers.

Example 4: Based on Sandwich Integration:

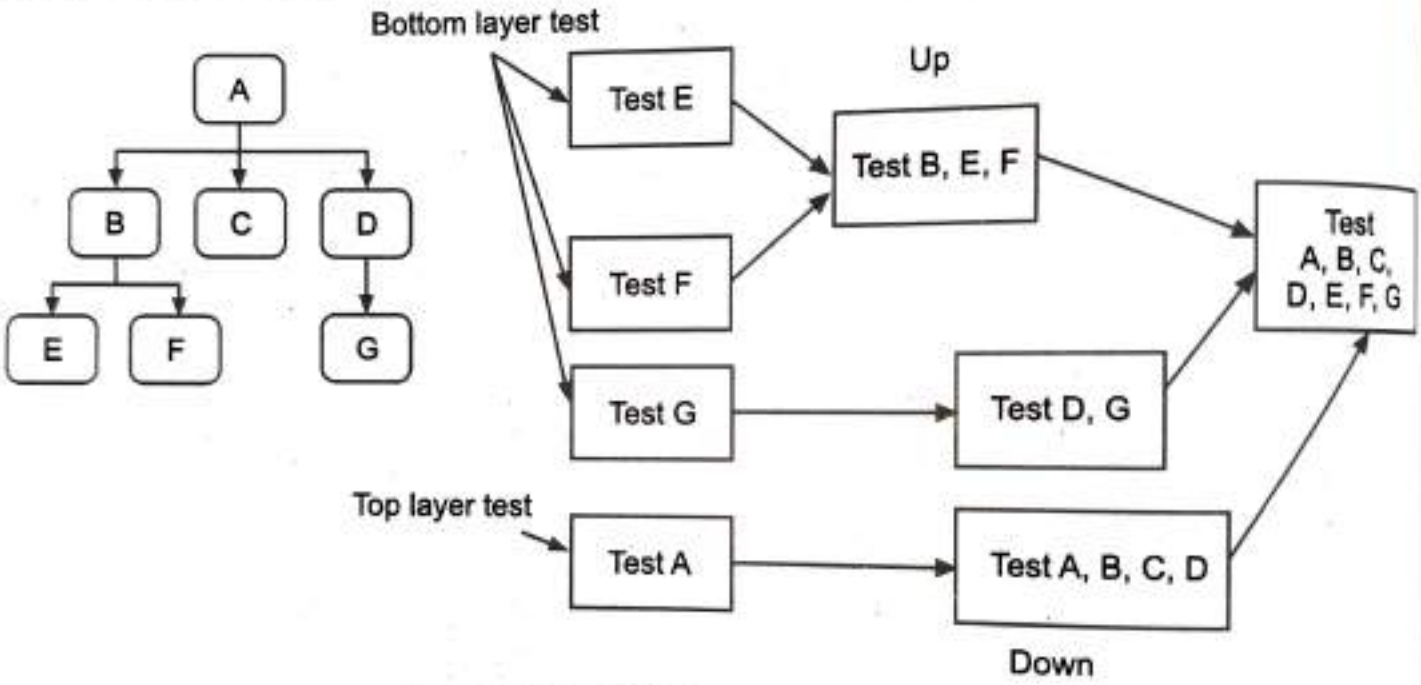


Fig. 3.13: Sandwich Integration Testing

Process of Sandwich Testing:

- The sandwich testing approach follows the following steps:

  Step 1: In this approach, bottom-up testing starts from middle layer and goes upward to the top layer. Generally, the bottom-up approach starts at a sub-system level and goes upwards for very big systems.

  Step 2: In sandwich testing, a top-down testing start from the middle layer and goes downward. Generally, the top-down approach starts at the subsystem level and goes downwards for very big system.

  Step 3: In this approach, the big-bang approach is followed for the middle layer. From this layer, bottom-up approach goes upwards and top-down approach goes downwards.

Advantage of Sandwich Integration Testing:

1. In this integration, both top-down and bottom-up approaches start at a time as per the development schedule. Units are tested and brought together to make a system. Integration is done downwards.
2. This approach is useful for very large projects having several sub-projects. When development follows a spiral model and the module itself is as large as a system.

Disadvantages of Sandwich Integration Testing:

1. It cannot be used for smaller systems with huge interdependence between different modules.
2. It represents a very high cost of testing as a lot of testing is done.
3. Does not test the individual subsystems thoroughly before integration.

3.10 PERFORMANCE TESTING

- Performance testing is used to uncover performance problems that can result from lack of server-side resources, inappropriate network bandwidth, inadequate database capabilities, faulty or weak operating system capabilities, poorly designed WebApp functionality, and other hardware or software issues that can lead to degraded Client-Server performance.
- The goal of performance testing is two fold:
  1. To understand how the system responds to loading (i.e. number of users, number of transactions, or overall data volume).
  2. To collect metrics (factors) that will lead to design modifications to improve performance.

3.10.1 Factors of Performance Testing

- Several factors that manage performance testing are:

1. **Throughput:**

- The capability of the system or the product in handling multiple transactions is determined by a factor of throughput. It represents the number of requests/transactions processed by the product in specified time duration.
- The throughput (that is, the number of transactions serviced by the product per unit time) varies according to the load the product is subjected to. Fig. 3.14 shows an example of the throughput of a system at various load conditions. The load to the product can be increased by increasing the number of users or by increasing the number of concurrent operations of the product.

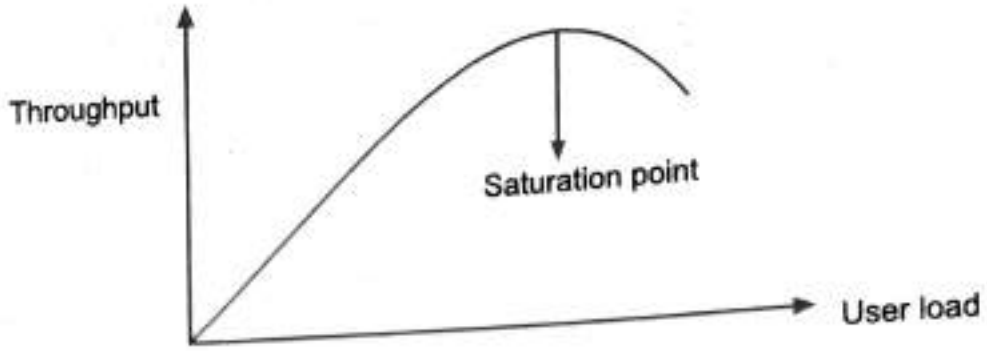


Fig. 3.14: Throughput of a System

- In the above example, it can be noticed that initially the throughput keeps increasing as the user load increases. This is the ideal situation for any product. It indicates that the product is capable of delivering more when there are more users trying to use the product.
- In the second part of the graph, it can be noticed that the throughput comes down. This is the period when users of the system notice a lack of satisfactory response and the system starts taking more time to complete business transactions. The "optimum throughput" is represented by the saturation point and it represents the maximum throughput for the product.

2. **Response Time:**

- Response time is another important activity of performance testing. It can be defined as the delay between the point of request and the first response from the product.
- In a typical Client/Server environment, throughput represents the number of transactions that can be handled by the server and response time represents the delay between the request and response.

3. **Latency:**

- In the networking scenario, the network or other products, which are sharing the network resources, can also cause delays. Hence, it is important to understand the delay caused by the product and the delay caused by the environment. Latency is a delay caused by the application, operating system, and the environment.
- Let us consider an example, where latency can be calculated for the product (database server, web server, etc) and for the network that represents the infrastructure available for the product. In such a scenario, if the network latency is more relative to the product latency and if that is affecting the response time, then we need to improve the network infrastructure. In other cases where network latency is more or cannot be improved, then we need to improve the product by using intelligent approaches of caching and sending multiple requests in one packet and receiving responses as a bunch.

4. **Tuning:**

- Tuning is a procedure by which the product performance is enhanced by setting different values to the parameters of the product, operating system, and other components.

5. **Benchmarking:**

- It is related to performance of competitive products. Performance of a product should match the performance of competitive products. Hence, it is very important to compare the throughput and response time of the product with those of the competitive products. This type of performance testing is called as Benchmarking.

6. **Capacity Planning:**

- One of the most important factors that affect performance testing is the availability of resources. A right kind of configuration is needed to derive the best results from performance testing. The procedure to find out what resources and configurations are needed is called capacity planning.
- The purpose of capacity planning is to help customer's plan for the set of hardware and software resources prior to installation or upgrade of the product.
- To summarize, performance testing is done to ensure that a product:
  - Processes the required number of transactions in any given interval (throughput).
  - Is available and running under different load conditions (availability).
  - Responds fast for different load conditions (response time).
  - Is comparable to and better than that of the competitors for different parameters (benchmarking).

3.10.2 Performance Testing Objectives [S-22]

- Performance tests are designed to simulate real-world loading situations. It may be the number of simultaneous users grows, or the number of on-line transactions increases, or the amount of data (downloaded or uploaded) increases.
- Performance testing will help to answer the following questions:
  - At what point (in terms of users, transactions, or data loading) does performance become unacceptable?
  - What system components are responsible for performance degradation?
  - What is the average response time for users under a variety of loading conditions?
  - Does performance degradation have an impact on system security?
  - Is WebApp reliability or accuracy affected as the load on the system increases?
  - What happens when loads that are greater than maximum server capacity are applied?

- To answer these questions, two different performance tests are conducted:
  1. *Load testing*: Real world loading is tested at a variety of load levels and in a variety of combinations.
  2. *Stress testing*: Loading is increased to the breaking point to determine how much capacity the WebApp environment can handle.
- For example, consider an application which can handle 1000 parallel users at a time. In load testing, the application is tested for 1000 users; whereas in stress testing, we test the application with more than 1000 users and the test continues to check where the application cracks. Therefore, stress testing is said to be as negative testing.

1. Load Testing:

- Load Tests are end-to-end performance tests under anticipated production load. The primary objective of this test is to determine how the application and its server-side environment will respond to various loading conditions.
- The test also measures the capability of the application to function correctly under load, by measuring transaction pass/fail/error rates.
- As testing proceeds, permutations to the following variables define a set of test conditions:
  - N, the number of concurrent users.
  - T, the number of on-line transactions per user per unit time.
  - D, the data load processed by the server per transaction.
- In every case, these variables are defined within normal operating bounds of the system. As each test condition proceeds, the following measures are collected:
  - Average user response, average time to download a standardized unit of data, or average time to process a transaction.
- The Web engineering team examines these measures to determine whether a decrease in performance can be traced to a specific combination of N, T and D.
- Load testing can also be used to measure recommended connection speed for users of the WebApp. Overall throughput, P, is computed in the following manner:

$$P = N \times T \times D$$

- As an example, consider a popular sports news site. At a given moment, 20,000 concurrent users submit a request once every two minutes on average (i.e. 0.5 per minute). Each transaction requires the WebApp to download a new article that averages 3 Kb in length. Therefore, throughput can be calculated as:

$P = [20{,}000 \times 0.5 \times 3\ \text{Kb}]/60 = 500\ \text{Kbytes/sec}$

$= 4\ \text{Mbps}$

- The network connection for the server would therefore have to support this data rate and should be tested to ensure that it does.
    - Load testing must be executed on "today's" production size database, and optionally with a "projected" database. If some database size increases in some month's time, then Load testing should also be conducted against a projected database.
    - It is important that such tests are repeatable as they need to be executed several times in the first year of wide scale deployment, to ensure that new releases and changes in database size do not affect the response time beyond the expectations.

2. **Stress Testing:**

- Stress Tests determine the load under which a system fails, and how it fails. Stress testing is a continuation of load testing, but in this instance the variables, N, T and D are forced to meet and then exceed operational limits.
- The intent of these tests is to answer each of the following questions:
    - Does the system degrade "gently" or does the server shut down as capacity is exceeded?
    - Does server software generate a "server not available" message? In general, are users aware that they cannot reach the server?
    - Are transactions lost as capacity is exceeded?
    - Is data integrity affected as capacity is exceeded?
    - What values of N, T and D force the server environment to fail?
    - If the system does fail, how long will it take to come back in-line?
- A variation of stress testing is sometimes referred to as **spike/bounce testing**. In this testing, load is spiked to capacity, then lowered quickly to normal operating conditions, and then spiked again. By bouncing system loading, a tester can determine how well the server can organize the resources to meet very high demand and then release them when normal conditions reappear.

3.10.3 Tools for Performance Testing

- There are two types of tools that can be used for performance testing – Functional performance tools and Load tools.
- Functional performance tools help in recording and playing back the transactions and obtaining performance numbers. This test involves very few machines.
- Load testing tools simulate the load condition for performance testing without knowing the number of users or machines.

1. Functional performance tools:
    - WinRunner
    - QA partner
    - Silk test

2. Load testing tools:
   - Load Runner
   - QA Load
   - Silk Performer

3.11 REGRESSION TESTING

- Whenever software is corrected, some aspect of software configuration is changed. Regression testing is the activity that helps to determine whether changed component has introduced any error in unchanged components.
- In other words, before a new version of a software product is released, the old test cases are run against the new version to make sure that all the old capabilities still work.

Definitions: [W-22; S-23]

- Regression means retesting a previously tested program to ensure that faults not have been introduced as a result of the changes made.
- Re-running previously conducted tests to ensure that unchanged components function correctly.
- Regression testing is a style of testing that focuses on retesting after changes are made. In traditional regression testing, we reuse the same tests. In risk-oriented regression testing, we test the same areas as before, but we use different (complex) tests.
- In Integration testing, adding a new module or changing the existing module impacts the system as:
  - New data flow paths are established.
  - New I/O may occur.
  - New control logic is invoked.
- These changes may cause problems with functions that previously worked correctly.
- In the context of integration test strategy, Regression testing is:
  "Re-execution of a subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects".

3.11.1 Regression Approaches

- It may be conducted by: [W-22; S-23]
  - **Manual testing:** By re-executing a subset of all test cases.
  - **Automated Capture/Playback Tools:** Capture the operation of an existing version of a system under test, and then the operation can be played back automatically on later versions of the system and any differences in behavior are reported.

3.11.2 Regression Test Suite

- Regression test suite (subset of tests to be executed) contains the following classes of test cases:
  - Representative sample of tests that exercises all software functions.
  - Additional tests that focus on functions that are affected by change.
  - Tests that focus on components that have been changed.

3.11.3 Types of Regression Testing

- When a test team or customer begins using a product, they report defects. These defects are examined by developer who makes individual defect fixes. The developer then does appropriate unit testing and checks the defect fixes into a Configuration Management system. The source code for the complete product is then compiled and these 'defect fixes' along with the existing features get merged into the build. Thus a build becomes an aggregation of all the 'defect fixes' and features that are present in the product.
- There are two types of regression testing:
  1. **Regular Regression Testing:** It is done between test cycles to ensure that the defect fixes are done and the functionality that was working with the earlier test cycles continue to work.
  2. **Final Regression Testing:** It is done to validate the final build before release. The Configuration Management engineer delivers the final build and other contents exactly as it would go to the customer.
- The final regression test cycle is conducted for a specific period of duration, which is mutually agreed between the development and testing teams.
- The final regression is more critical than any other type of testing, as this is the only testing that ensures the same build of the product that reaches to the customer.

3.11.4 When to do Regression Testing?

- Whenever changes happen to software, regression testing is done to ensure that these do not adversely affect the existing functionality.
- It is necessary to perform regression testing when:
  1. A reasonable amount of initial testing is already carried out.
  2. A good number of defects have been fixed.
  3. Defect fixes that can produce side effects are taken care of.
- When a developer fixes a defect, the defect is sent back to the test engineer for verification. The test engineer needs to take the appropriate action of closing the defect if it is fixed, otherwise reopen it if it has not been fixed properly. In this process, regression testing needs to be done to avoid side effects. To ensure that there

are no side effects, some more test cases are selected and defect fixes are verified in the regression test-cycle.

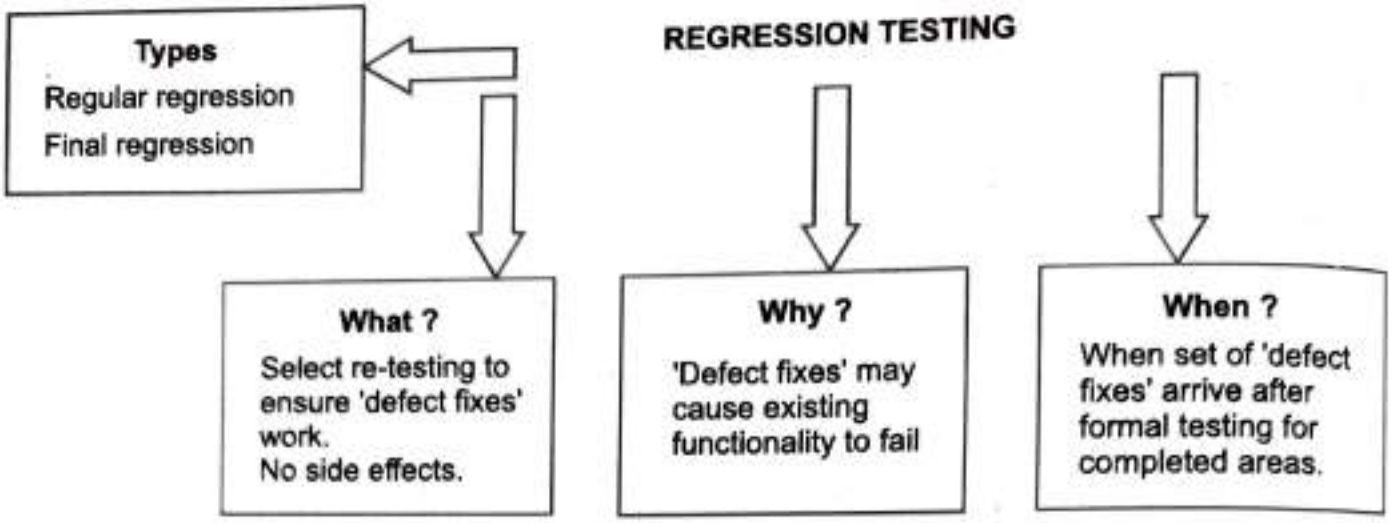


Fig. 3.15: Concept of Regression Testing

- Regression testing differs from retesting. Retesting means only the certain part (module) of an application is tested again without considering how it will affect the other part of application. Whereas, in Regression Testing the entire application is tested after a change in a module or part of the application, to ensure that the code change does not affect existing functionality of the application.

3.11.5 How to do Regression Testing?

- There are several methodologies for regression testing that are used by different organizations. A methodology that includes by the majority of them is made of the following steps:
  - Perform an initial Smoke test.
  - Understand the criteria for selecting the test cases.
  - Classify the test case into different priorities.
  - Select test cases.
  - Resetting the test cases for execution.
  - Conclude the results of the regression cycle.

1. **Perform an initial Smoke test (Covered in section 3.12)**
2. **Understand the criteria for selecting the test cases:**

- There are two approaches for selecting the test cases.
  1. An organization can choose to have a constant set of regression tests that are run for every build or change. But this approach is not always feasible because:
     - In order to cover all fixes, the "constant set of tests" will include all features and tests which are not required, may execute every time.

- A given set of defect fixes or changes may introduce problems for which there may not be ready test cases in the constant set.

2. A second approach is to select the test cases dynamically for each build. Such selection of test cases requires knowledge of:
   - The defect fixes and changes made in the current build.
   - The ways to test the current changes.
   - The impact that the current changes may have on other parts of the system.
   - What to test on affected parts?

Some of the criteria to select test cases are as follows:

(i) Include test cases that have produced the maximum defects in the past.

(ii) Include test cases that test the basic functionality or the core features of the product, which are mandatory requirements of the customer.

(iii) Include test cases that test the end-to-end behavior of the application.

(iv) Include test cases to test the positive test conditions.

Classify the test case into different priorities:

It is important to know the relative priority of test cases for a successful test execution. Test cases can be classified into three categories:

- **Priority-0:** These test cases check **basic functionality** and are run to accept the build for further testing. They are also executed when a product goes through a major change. These test cases deliver very high project value to both the development team and to the customers.
- **Priority-1:** These test cases use the **basic and normal** set-up. These test cases deliver high project value to both the development team and to the customers.
- **Priority-2:** These test cases deliver **moderate** project value. They are executed as a part of the testing cycle and are selected for regression testing on a need basis.

Classification of the above test case priorities is illustrated in Fig. 3.16.

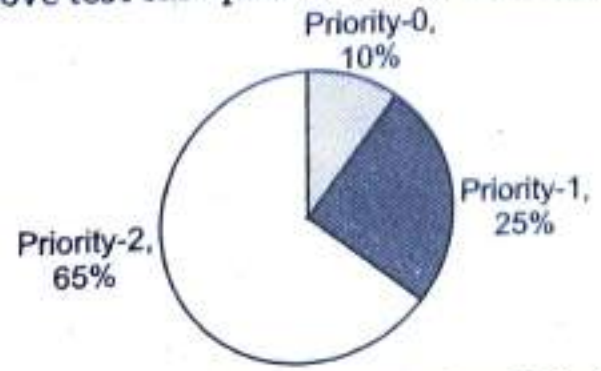


Fig. 3.16: Classification of Test Case Priority

4. Select Test Cases:

- Once the test cases are classified into different priorities, then test cases can be selected. The methodology discussed below takes into account the **criticality** and **impact of defect fixes** after test cases are classified into several priorities.
  - **Case 1:** If the criticality and impact of the defect fixes are **low**, then the test engineer selects few test cases from Test Case Database (TCDB) (a repository that stores all the test cases). These test cases can fall under any priority (0, 1, or 2).
  - **Case 2:** If the criticality and impact of the defect fixes are **medium**, then the test engineer selects all Priority-0 and Priority-1 test cases. If defect fixes need additional test cases, then Priority-2 test cases are also selected for regression testing.
  - **Case 3:** If the criticality and impact of the defect fixes are **high**, then the test engineer selects all Priority-0, Priority-1 and a carefully selected subset of Priority-2 test cases.
- Above methodology requires analysis of criticality and the impact of defect fixes. This can be time consuming. Alternative methodologies can be:
  - **Regress All:** All priority 0, 1, and 2 test cases are executed. This means **all** the test cases in the regression test bed/test-suite are executed.
  - **Priority-based Regression:** All priority 0, 1, 2 test cases are run **in sequence** based on the availability of time.
  - **Random Regression: Random** test cases are selected and executed for regression methodology.
  - **Context-based Dynamic Regression:** Few priority-0 test cases are selected, and based on the context and outcome; additional related test cases are selected for continuing the regression testing.

5. Resetting the Test Cases for execution:

- After selecting the test cases, the next step is to prepare the test cases for execution.
- In a large product release involving several rounds of testing. It is very important to record what test cases were executed in which cycle, their results and related information. This is called *test case result history*.
- A method or procedure that uses test case result history to indicate some of the test cases, which are selected for regression testing, is called a *reset procedure*. Resetting a test case is nothing but setting a flag called not run or execute again in the test case DB.

6. Conclude the results of the Regression cycle:

- Regression uses test cases that have been already executed more than once. So, it is expected that 100% of those test cases pass the test using the same build. On the other

hand, where the pass percentage is not 100, the test manager can compare current results with the previous results of the test case to conclude whether regression was successful or not.

- If the result of a particular test case was a PASS using the previous builds and a FAIL in the current build, then regression has FAILED. A new build is required and the testing must start from scratch after resetting the test cases.
- If the result of a particular test case was a FAIL using the previous builds and a PASS in the current build, then regression has PASSED. Here it is safe to assume that the defect fixes worked.
- If the result of a particular test case was a FAIL using the previous builds and a FAIL in the current build and if there are no defect fixes, then regression has FAILED. This means such test cases should not be selected for regression.
- If the result of a particular test case was a FAIL using the previous builds but works with a documented WORKAROUND and if you are satisfied with the workaround, then it should be considered as a PASS for both the system and regression test cycle.
- If you are not satisfied with the WORKAROUND, then it should be considered as a fail for a system test but may be considered as a pass for a regression test cycle.

Above discussed conclusions are summarized in Table 3.3.

Table 3.3: Results of a Regression Test Cycle

Previous result	Current result	Conclusion	Remarks
PASS	FAIL	FAIL	Need to improve the regression process and code reviews.
FAIL	PASS	PASS	This is the expected result of a good regression.
FAIL	FAIL	FAIL	Need to analyze why defect fixes are not working.
FAIL	PASS (with a workaround)	PASS (if satisfied with workaround)	Workarounds also need a good review as they can also create side-effects.
PASS	PASS	PASS	There are no side-effects due to defect fixes.

3.12 SMOKE TESTING

[W-22; S-23]

- A smoke test is reduced version of regression testing. Smoke testing is the initial testing process exercised to check whether the software under test is ready for further processing.
- The term smoke testing originated in the hardware industry. After a part of hardware or a hardware component (For example: transformer) was changed or repaired, the equipment was simply powered-up. If there was no smoke, the component passed the test.
- In software, the term *smoke testing* describes the process of validating and confirming the availability of changed code to identify and fix defects. After this test, the changes are mapped into the source product. It is designed to confirm that changes in the code functions as expected.
- Before running the smoke test, code review is conducted which focuses on changes in the code. Tester must work with the developer who has written the code to understand:
  - What changes are made in the code?
  - How does the change affect the functionality?
  - How does the change affect the interdependencies of various components?
- Whenever a new software build is received, a smoke test is conducted against the software, verifying that the major functionality still operates. Smoke testing consists of:
  - Identifying the basic functionality that a product must satisfy.
  - Designing test cases to ensure that these basic functions work and then packaging them into a smoke test suite.
  - Ensuring that every time a product is built, this suite runs successfully.
  - If this suite fails, then developers should identify the changes and possibly change or roll back these changes to a state where the test suite succeeds.
- McConnell describes the smoke test as:
  - "The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems".
- This type of testing is applicable in the Integration Testing, System Testing and Acceptance Testing levels.

Advantages of Smoke testing:

1. It helps to find issues introduced in integration of modules and in the early phase of testing.

2. It helps to get confidence to tester that fixes done in the previous build is not affecting major features.

3.13 LOAD TESTING [W-22; S-23]

- Load testing is a software testing technique used to examine the behavior of a system under the normal and extreme expected load conditions. This testing is generally performed under controlled laboratory conditions in order to distinguish between two different systems. This is a type of non-functional testing.
- This testing is performed to determine a system's behavior under both normal and at peak conditions. It helps to identify the maximum operating capacity of an application as well as any bottlenecks and determine which element is causing degradation. For example, if the number of users are increased then how much CPU, memory will be consumed, what is the network and bandwidth response time.
- This involves simulating real-life user load for the target application. It helps you determine how your application behaves when multiple users work simultaneously.
- Load testing differs from stress testing, which evaluates the extent to which a system keeps on working under extreme workloads or when some of its hardware or software has been compromised. The primary goal of load testing is to define the maximum amount of work a system can handle without performance degradation.

Examples of Load Testing:

1. Downloading a series of large files from the Internet.
2. Running multiple applications on a computer or server simultaneously.
3. Assigning many jobs to a printer in a queue.
4. Subjecting a server to a large amount of traffic.
5. Writing and reading data to and from a hard disk continuously.

Advantages of Load Testing:

1. Load testing will expose the bugs such as undetected memory overflow and memory management bugs in your system.
2. Load testing enables you to increase your uptime in your internet system. Load testing can do this because it spots the bottlenecks even before they happen in an actual environment.
3. It can measure the performance of your internet infrastructure. For example, if you are engaged in e-commerce, then you can monitor how your business is doing especially, when there are many concurrent users who hit your site.
4. Load testing will also prevent software failures, because it can predict how the system will react when it is given large loads of files and a large amount of tasks.

5. Investments can be protected, because this kind of testing allows investors to understand the scalability and the performance of delivered software.

Summary

- In this chapter, we have learned various testing levels and testing strategies.
- Once the code is ready then the system module passes through various testing levels dynamically, which starts with unit testing. Every module testing is done against the defined structure and defined functionalities.
- Various aspects of structural and functional testing are discussed in this chapter.
- Concepts of stubs and drivers are explained in the context of unit and integration testing.
- Driver is required for missing caller modules and Stubs are required for missing called modules.

Check Your Understanding

1. A _____ is a piece of software which calls the functions in the unit under test.
   (a) Stub (b) Driver
   (c) Procedure (d) Function
2. _____ are programs that are used to replace modules that are subordinate to the module to be tested.
   (a) Stub (b) Driver
   (c) Procedure (d) Function
3. Which is called as module level testing?
   (a) System level (b) Unit level
   (c) Design level (d) Function level
4. Acceptance testing is done to check _____.
   (a) Client Needs are met (b) Structure is correct
   (c) Design specifications are met (d) Requirements are met
5. Integration testing is done for _____.
   (a) System validation (b) System acceptance
   (c) Requirement testing (d) Domain level testing
6. Alpha and beta are _____.
   (a) Acceptance testing types (b) System testing types
   (c) Unit testing types (d) Regression testing types
7. _____ testing is performed under controlled laboratory conditions in order to distinguish between two different systems.
   (a) Smoke (b) System
   (c) Load (d) Regression
8. 'Smoke testing is a kind of regression testing', True/False?
   (a) True (b) False
   (c) May be (d) Not Necessary

9. Which of the following is NOT a Functional performance tool?
   (a) Winrunner
   (b) LoadRunner
   (c) QTP
   (d) All of the Above
10. Which is the correct statement for describing the testing levels?
   (a) Unit, integration, System, Acceptance.
   (b) Integration, System, Unit, Performance.
   (c) Component, System, Performance, Validation.
   (d) Module, Unit, Integration, System.

Answers

1. (b)	2. (a)	3. (b)	4. (a)	5. (d)
6. (a)	7. (c)	8. (a)	9. (b)	10. (a)

Practice Questions

Q.I Answer the following questions in short.

1. What is Validation Testing technique? Explain with an example.
2. Write a short note on Agile testing, Load testing.
3. What is System testing in Software Testing?
4. What is a Regression testing?
5. Explain in short a concept of Complexity Metrics.
6. What is the Verification concept in software testing?
7. What are types of Regression testing?
8. Explain Sandwich approach of testing.
9. Explain types of Acceptance testing?

Q.II Answer the following questions.

1. Explain Unit testing with an example.
2. Why do we need Integration testing? Explain top-down and bottom-up integration testing in details.
3. What do you mean by stub and driver? Explain with an example.
4. Explain testing strategies in detail.
5. Differentiate Validation and Verification techniques.
6. Write a brief note on Smoke testing.
7. Why do we need Regression testing? Explain how Regression testing is done.
8. Differentiate Alpha and Beta level Acceptance testing.

Q.III Define the terms.

1. Big Bang Approach
2. Top-down Integration
3. Beta Testing
4. Stub
5. Module Testing
6. Performance Testing
7. Smoke Testing

4... Software Metrics

Learning Objectives ...

- To learn about Software Metrics.
- To get knowledge of basic metrics such as size-oriented metrics and function-oriented metrics.
- To learn about Cyclomatic Complexity Metrics and its examples.

4.1 INTRODUCTION TO METRICS

- A Metric is a quantitative measure of the degree to which a system, system components, or process possesses a given attribute. Software metrics play an important role in measuring attributes that are critical to the success of a software project. Measurement of these attributes helps to make the characteristics and relationships between various attributes. This is helpful in decision-making.
- Software measurements previously were limited to measuring individual, product software attributes. Rather than the one-dimensional approach towards measurements, organizations are coming forward with integrating complete software metric programs into the software development processes. Measuring the software metrics is not only for process improvements but also to reach higher Capability Maturity Model levels.

Properties of the Metrics:

- **Simple, definable:** The metrics should be simple and definable so that we get a clear idea about how the metrics can be evaluated.
- **Objective:** The metrics should be objective to the greatest extent possible. The metrics should be easily available that is the cost to get the metrics should be reasonable (i.e. related to cost).
- **Valid:** The metric should measure what is intended to measure.
- **Robust:** The metric should be relatively insensitive to insignificant changes in the process or product.

Type of Metrics:

1. **Process Metrics:** It can be used to improve the process efficiency of the SDLC (Software Development Life Cycle).
2. **Product Metrics:** It deals with the quality of the software product.
3. **Project Metrics:** It can be used to measure the efficiency of a project team or any testing tools being used by the team members.

Table 4.1 Types of Metrics

Category of Metrics	Example	Metric
Process	Testing	Number of defects found
		Testing Time
Product	Program	Size
		Complexity
Resource	Personnel	Number of programs
		Skills

Types of Metric according to Measurements:

- In Software Engineering, Manual test metrics are classified into two classes:

1. **Base Metrics (Direct Measures):**

- Base metrics constitute the raw data gathered by a Test Analyst throughout the testing effort. These metrics are used to provide project status reports to the Test Lead and Project Manager; they also feed into the formulas used to derive Calculated Metrics.
- Example: # of Test Cases, # of Test Cases Executed

2. **Calculated Metrics (Indirect Measures):**

- Calculated Metrics convert the Base Metrics data into more useful information. These types of metrics are generally the responsibility of the Test Lead and can be tracked at many different levels (by module, tester, or project).
- Example: % Complete, % Test Coverage.

4.2 BASIC METRICS

Software measurements:

- Software measurements are of two types, namely, Direct measures and Indirect measures.
- Direct measures include software processes like cost and effort applied and products like lines of code produced, execution speed, and other defects that have been reported.
- Indirect measures include products like functionality, quality, complexity, reliability, maintainability, and many more.

1. **Size:**

- The simplest measure of software is its size. Two possible metrics are: size in bytes and size in the number of statements.
- The size in statements is often termed as LOCs (lines of code), sometimes SLOCs (source lines of code). The size in bytes affects the main memory and disk space requirements and affects performance. The size measured in statements relates to development effort and maintenance costs. But the large program does not necessarily take a longer time to develop than the small program, because the complexity of the software also causes some effect. A metric such as LOCs take no account of complexity. (We shall see shortly how complexity can be measured)
- There are different ways of interpreting even a simple metric like LOCs since it is possible to exclude or include, comments, data declaration statements, and so on. Blank lines are not included in the count.

2. **Person-months:**

- The second major metric is person-months, a measure of developer effort. Since people's time is the major factor in software development, person-months usually determine cost. If an organization measures the development time for components, the information can be used to predict the time of future developments. It can also be used to gauge the effectiveness of new techniques that are used.

3. **Bugs:**

- The third basic metric is the number of bugs. As the component is being developed and tested, a log can be kept of the found bugs. In week 1 there might be 27 bugs, in week 2 there might be 13 bugs, and so on. This may help in predicting how many bugs remain at the end of the development. These figures can also be used to assess how good new techniques are.

4.2.1 Software Metric

[S-22, 23; W-22]

- Software Metric is a measure of some property of a piece of software or its specifications. It is a term that contains many activities, all of which involves some degree of software measurement such as:

1. Cost and Effort estimation.
2. Productivity measures.
3. Data collection.
4. Quality models and Measures.
5. Performance evaluation.
6. Structural and Complexity metrics.
7. Management by metrics.
8. Capability-Maturity assessment.

1. **Cost and Effort Estimation:**

- Numerous models for software costs and effort estimation (i.e. COCOMO – Constructive Cost Model, SLIM model, etc.) have been proposed, to predict project costs during the early phases of the software lifecycle.
- These models often share a common approach: Effort is expressed as a function of one or more variables, such as the size of the product, the capability of the developers, and the level of reuse. Size is expressed in terms of Lines of codes or the number of function points.
- Test in-process predictions (estimations) needed for software development:
  - **Coding:** Size/Schedule/Quality predictions.
  - **Testing:** End of testing predictions.
  - Reliability/Quality predictions.

2. **Productivity Models and Measures:**

- Fig. 4.1 illustrates an example of the possible components that contribute to overall productivity.
- It shows productivity as a function of value and cost; each is then decomposed into other aspects, which are expressed in measurable form.

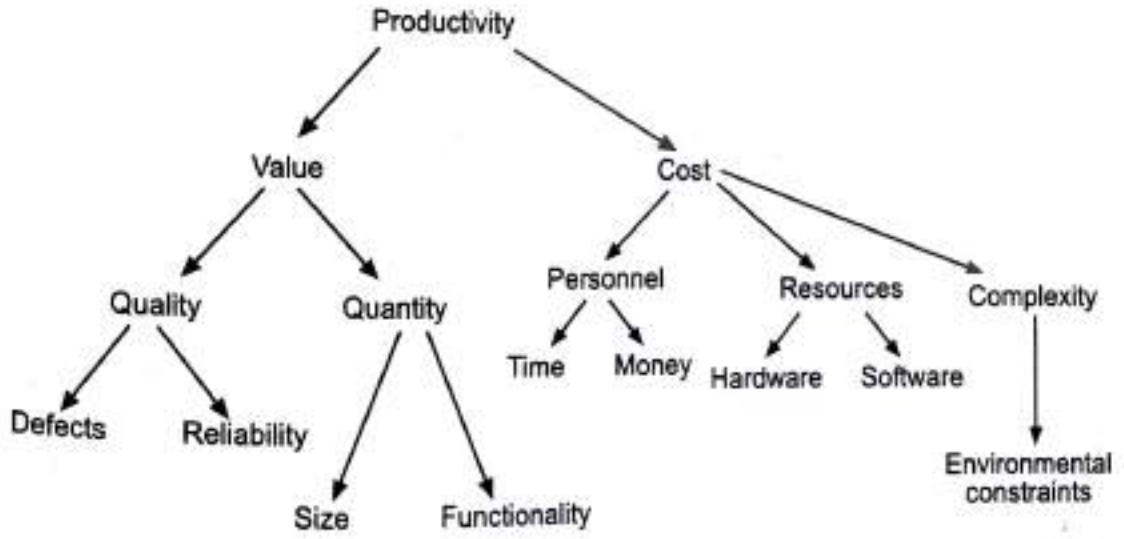


Fig. 4.1: A Productivity Model

3. **Data Collection:**

- The quality of any measurement program is dependent on careful data collection. Data collection is required to ensure that collection is consistent, complete and data integrity is not at risk.
- Fig. 4.2 contains several examples of data collection, which helps the managers to study the progress as well as problems of development.

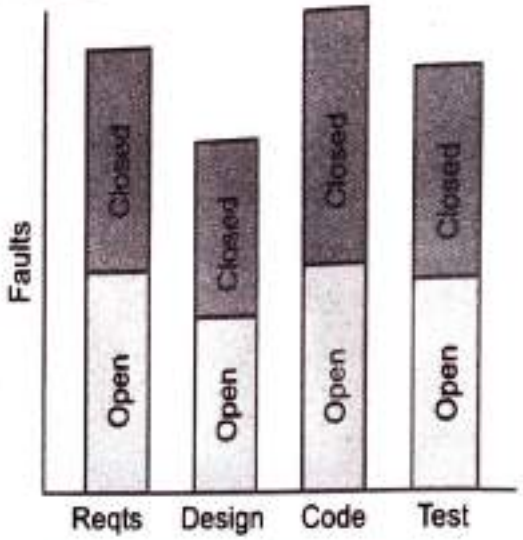


Fig. 4.2: Metrics for Management

4. **Quality Models and Measures:**

- These models are usually constructed in a tree-like fashion as shown in Fig. 4.3.

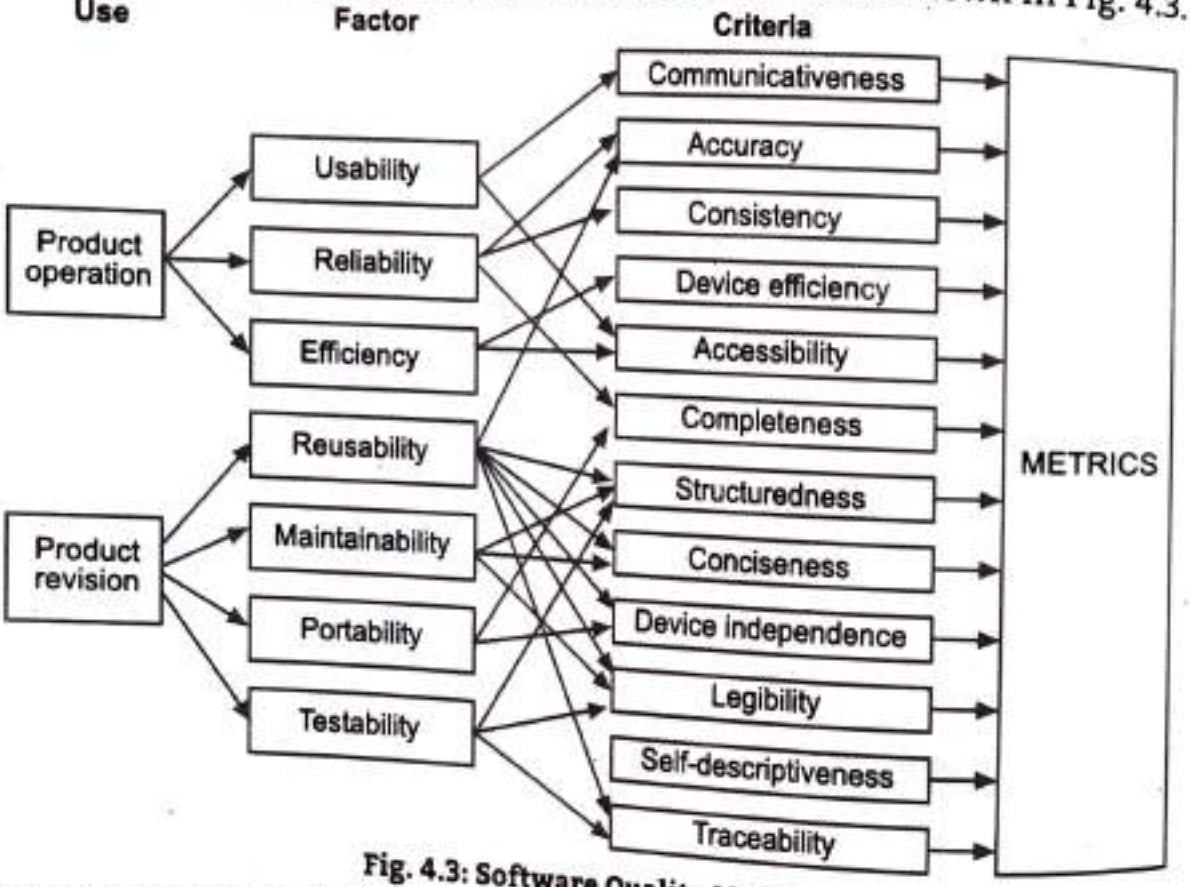


Fig. 4.3: Software Quality Model

- The upper branches hold important **high-level** quality factors of software products, such as reliability and usability. Each quality factor is composed of lower-level criteria such as structured ness and traceability.
- Tree describes the relationship between factors and their dependent criteria.

5. **Performance Evaluation and Models:**

- Performance is another important aspect of quality. Performance evaluation means observing system performance characteristics such as response times and completion rates.

- Performance investigation involves the efficiency of algorithms and algorithmic complexity.

6. **Structural and Complexity metrics:**

- The complexity of a problem is defined as the number of resources required for an optimal solution to the problem. The complexity of a solution is the resources needed to implement a particular solution.
- Complexity can be interpreted in the following ways:
  - **Problem Complexity:** Measures the complexity of the underlying problem.
  - **Algorithmic Complexity:** Measures efficiency of software by measuring Time and Space complexity.
  - **Structural Complexity:** Measures the structure of the software used to implement the algorithm.

  For example, Control Flow structure, Hierarchical Structure measures, McCabe's Cyclomatic complexity measures.

7. **Management by Metrics:**

- Measurement is an important part of software project management.
- Customers and developers rely on measurement-based charts and graphs to help them decide if the project is on track.

8. **Capability-Maturity Assessment:**

- Software Engineering Institute (SEI) proposed a Capability Maturity Model (CMM) to measure an organization's ability to develop quality software. The CMM describes an evolutionary improvement path from an ad-hoc, immature process (dependant on individuals) to a mature, disciplined process that could be optimized based on continuous feedback.

4.2.2 Capability Maturity Model (CMM) [S-22]

- The Capability Maturity Model had its beginning as the Process Maturity Model, where an organization was rated on an ordinal scale from 1 (low) to 5 (high), based on answers to the questions about its development process.
- The model is used in two ways:
  1. By customers, to identify the strengths and weaknesses of their suppliers.
  2. By software developers, to assess their capabilities and set a path toward improvement.
- The Capability Maturity Model for Software (CMM or SW-CMM) is an industry-standard model for defining and measuring the maturity of a software company's development process and for providing direction on what they can do to improve their software quality.

- The CMM covers practices for planning, engineering, and managing software development and maintenance. When followed, these key practices improve the ability of organizations to meet goals for cost, schedule, functionality, and product quality.

Definition of CMM Maturity Levels:

- CMM is a framework, which is composed of five maturity levels:

  Level 1: Initial.

  Level 2: Repeatable.

  Level 3: Defined.

  Level 4: Managed.

  Level 5: Optimizing.

- Except level 1, each maturity level is composed of key process areas on which an organization should focus as part of its improvement activities.
- Fig. 4.4 shows the five maturity levels of the CMM. Each maturity level provides a layer in the foundation for continuous process improvement.

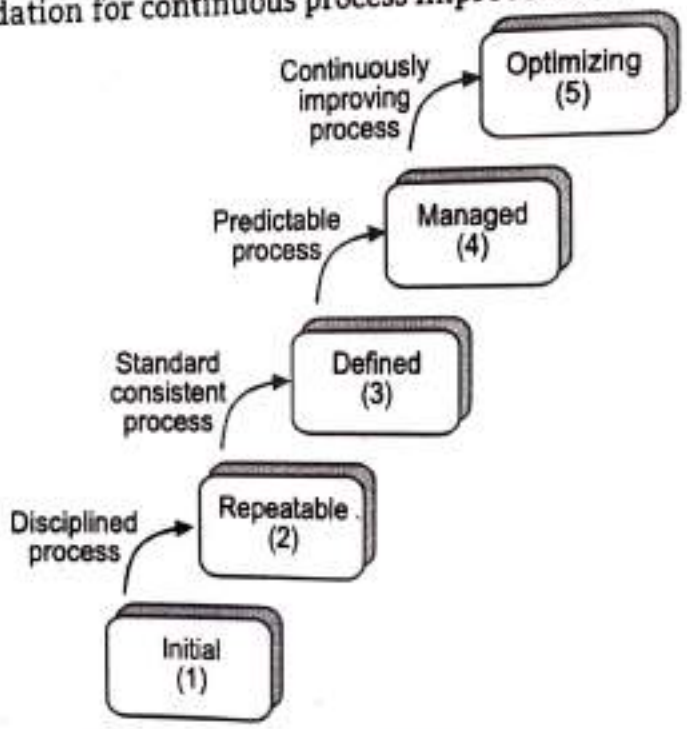


Fig. 4.4: Five Levels of Software Process Maturity

1. **Level 1- Initial - (Chaotic)**

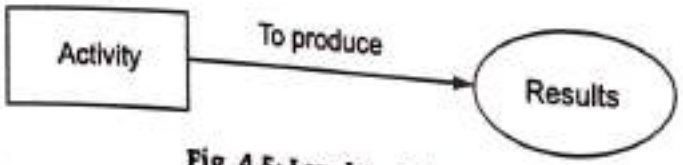


Fig. 4.5: Level 1 – Initial

- It works on the principle – "Just do it". No formal process is used for estimation and planning. Lack of management mechanism to ensure that processes are being used.

- The software process capability is unpredictable because the software development is ad-hoc and no well-defined processes are followed. Schedules, budgets, functionality, and product quality are generally unpredictable.
- Performance depends on the capabilities of individuals and varies with their skills, knowledge, and motivations.
- During the crisis, projects typically abandon planned procedures and revert to coding and testing. Success depends entirely on having an exceptional manager and an effective software team.

2. **Level 2 - The Repeatable Level:**

- Level 2 works on the principle – "Think before you act, and think after you act; just to make sure what you did is right!"

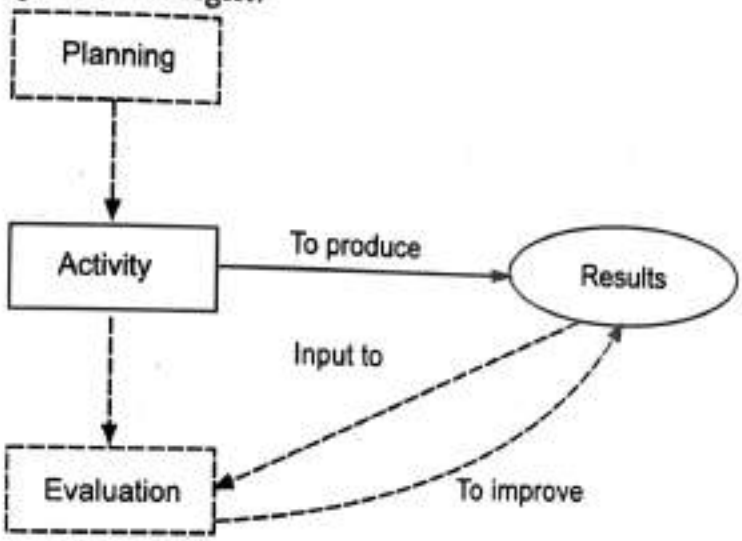


Fig. 4.6: Level 2 - Repeatable

- At the repeatable level, policies for managing a software project and procedures to implement those policies are established. Planning and managing new projects is based on experience with similar projects.
- The key idea behind achieving Level 2 is - To adopt effective management processes for software projects, which allow organizations to repeat successful practices developed on earlier projects.
- Project management processes - Track software costs, schedules, and functionality.
- The key process areas to achieve level 2 maturity are listed below:
  - Requirements management.
  - Software project planning and oversight.
  - Software subcontract management.
  - Software quality assurance.
- Software configuration management.

3. **Level 3 - The Defined Level:**

- Level 3 works on the principle – "Use your lessons learned".

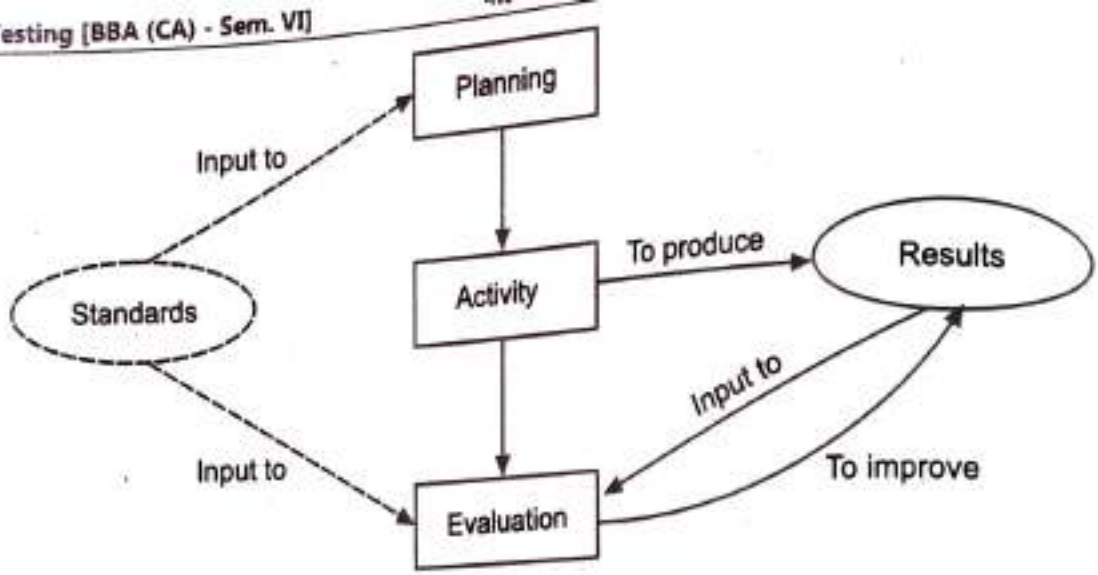


Fig. 4.7: Level 3 - Defined

- At the defined level, the standard process for developing and maintaining software across the organization is recognized, including both software engineering and management processes.
- Throughout the CMM this standard process is referred to as the organization's standard software process.
- An organization-wide training program is implemented to ensure that the staff and managers have the knowledge and skills required to fulfill their assigned roles.
- Projects adopt the organization's standard software process to develop their own defined software process, which accounts for the unique characteristics of the project. A defined software process contains a consistent, integrated set of well-defined software engineering and management processes.
- The key process areas to achieve level 3 maturity are listed below:
  - Organizational process improvement.
  - Organizational process definition.
  - Training program.
  - Integrated software management.
  - Software product engineering.
  - Intergroup co-ordination.
  - Peer reviews.

4. Level 4-The Managed Level:

- Level 4 works on the principle – "Predict the results you need or expect and then create opportunities to get those results".
- At the managed level, the organization sets quantitative quality goals for both software products and processes. An organization-wide software process database is used to collect and analyze the data available from the project's defined software processes.

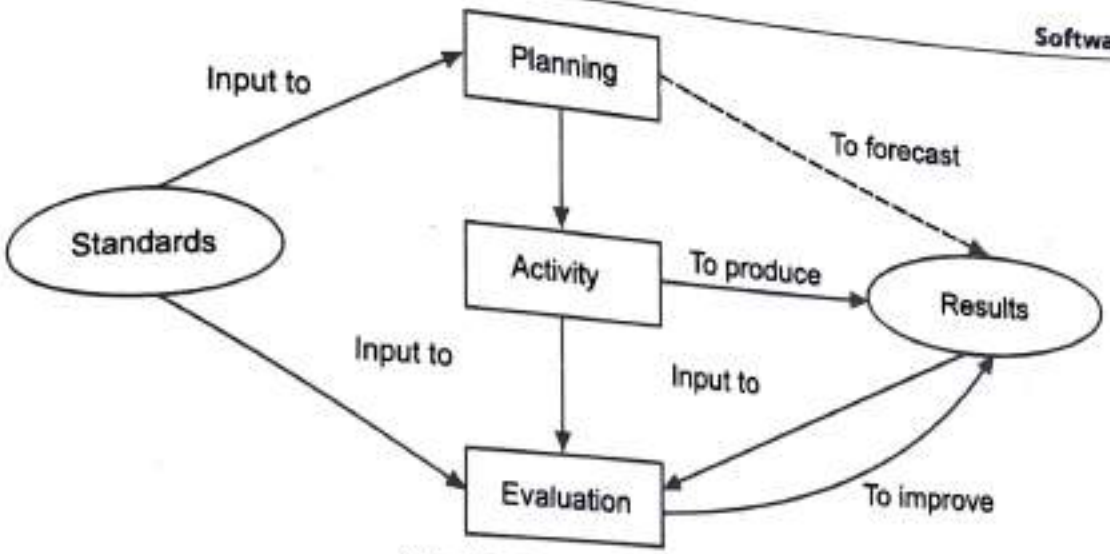


Fig. 4.8: Level 4 - Managed

- The software process capability of Level 4 organizations can be summarized as predictable because the process is measured and operates within measurable limits. When these limits are exceeded, action is taken to correct the situation. Software products are of predictably high quality. At this level, the entire software development process is not only defined but is managed proactively.
- The key process areas to achieve this level are listed below:
  - Process measurement and analysis.
  - Quality management.

5. **Level 5 - The Optimizing Level:**

- Level 5 works on the principle – "Create lessons learned, and use lessons learned to create more lessons, and use more lessons learned to create even more lessons, and so on..."

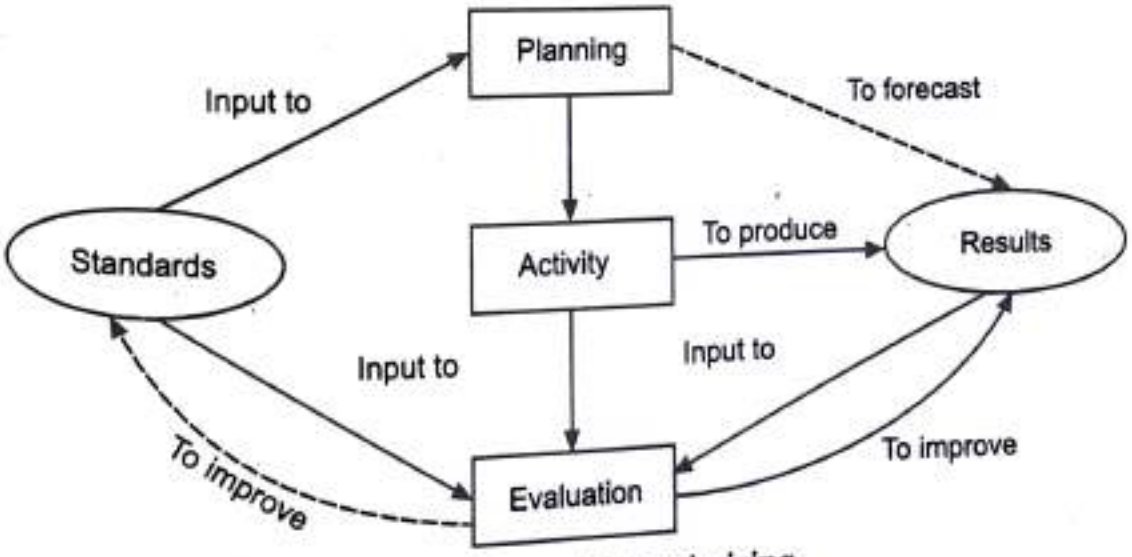


Fig. 4.9: Level 5- Optimizing

- At the optimizing level, the entire organization is focused on continuous process improvement with the goal of defect prevention.
- Software project teams analyze defects to determine their causes. Software processes are evaluated to prevent known types of defects from recurring.

- The software process capability of Level 5 organizations can be characterized as continuously improving because Level 5 organizations are continuously striving to improve the process performance. Very few large organizations have ever achieved a level 5 score in SEI evaluations.
- The key process areas to achieve this level are listed below:
  - Defect prevention.
  - Technology innovation.
  - Process change management.
  - Quality management.
- The software processes of the SW-CMM can be applied across the entire software lifecycle, from requirements gathering through final testing.
- Fig. 4.10 summarizes the maturity levels and Key Process Areas (KPA's).

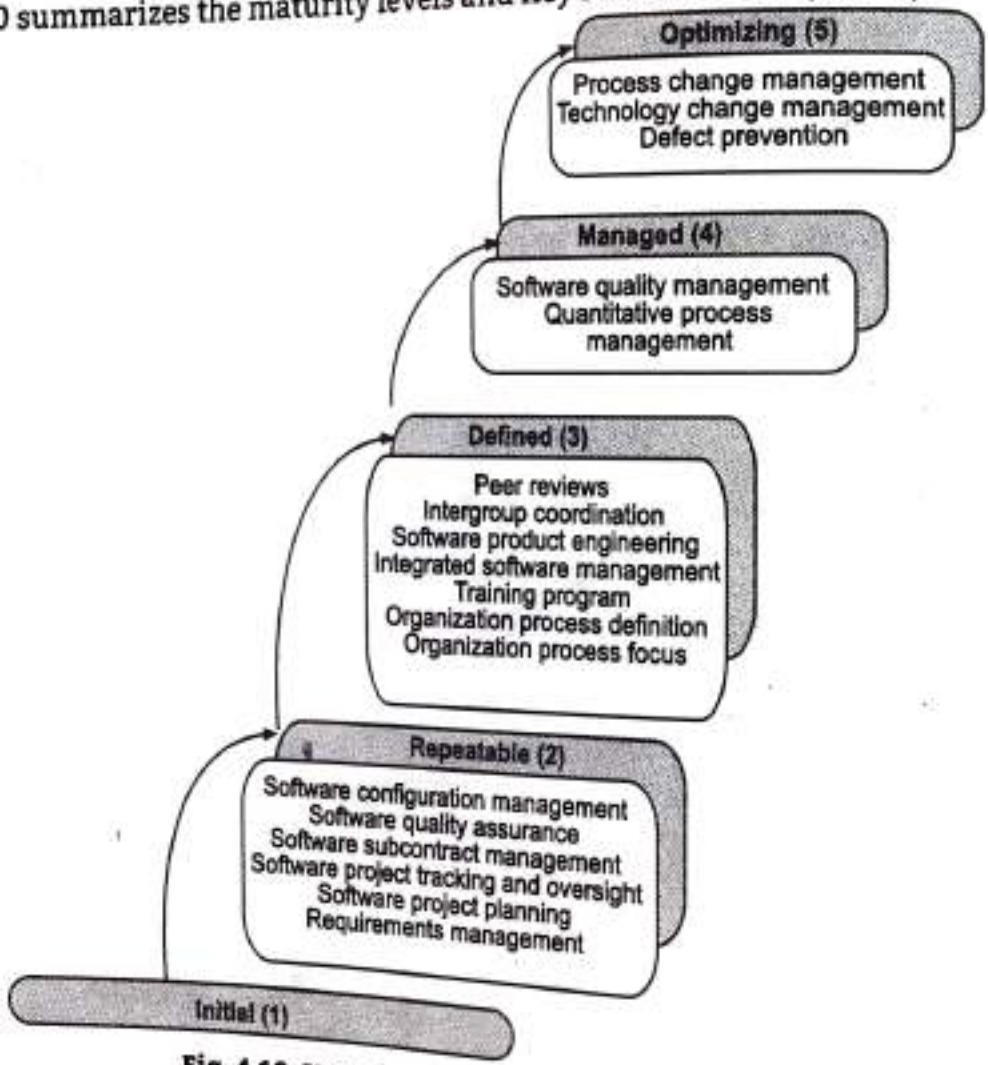


Fig. 4.10: Key Process Areas by Maturity Level

4.2.3 Metrics Plan

- As discussed earlier, software metric is a measure of some property of a piece of software or its specifications. It is a term that contains many activities, all of which involve some degree of software measurements, such as cost and effort estimation, productivity measures, data collection, and others.

- A metrics plan must describe - who, what, where, when, how, and why of metrics i.e. 5 W's and 1 H.
- The plan begins with why, laying out the goals or objectives of the project, describing what questions need to be answered by project members and project management. For example, If reliability is a major concern, then the plan discusses:
  - How reliability will be defined?
  - What is the reliability requirement?
  - What are the timescales?
- On the other hand, if we consider productivity as a major concern, then the plan discusses:
  - How to improve the productivity of the team?
  - What is the team's productivity?
  - What is the quality of the product?
- Next, the plan addresses what will be measured. For example, Reliability can be measured in terms of size, failures and defects discovered, etc. Productivity may be measured in terms of product size, effort, team size, and other parameters.
- In this section, the metrics plan will explain how the size, defects, and efforts are defined along with how they are combined to compute reliability or productivity. At the same time, the plan must lay out where and when during the process the measurements will be made.
- How and who addresses the identification of tools, techniques, and staff available for metrics collection and analysis. The plan must state clearly what types of analysis will be carried out with the data, who will do the analysis, how the results will be conveyed to the decision-makers, and how they will support decisions. The metrics plan must address the people who perform each of these tasks on data:
  - Capturing.
  - Formatting.
  - Validating.
  - Analyzing.
  - Presenting.
  - Using (within the goals of the program).
  - Evaluating (effectiveness of the metrics plan).
- Thus, the metrics plan paints a complete picture of the measurement process from the initial definition of need to analysis and application of results.

4.2.4 Goal-Question-Metric Model (GQM)

- The Goal/Question/Metric (GQM) Paradigm developed by Basili and Weiss is a mechanism that provides a framework for developing a metrics program.
- The GQM paradigm consists of three steps:
  1. Generate a set of goals for the development or maintenance project.
  2. Derive from each goal the questions that must be answered to determine if the goals are being met.
  3. Decide what must be measured (a set of metrics) to be able to answer the questions adequately.
- Let us see each step in detail.

1. Generate a set of goals based upon the needs of the organization:

- Determine what you want to improve, evaluate or examine. Goals are defined in terms of purpose, perspective, and environment using generic templates:
  - **Purpose:** To (characterize, evaluate, improve, predict, motivate, etc.) the (process, product, model, metric, etc.) to (understand, assess, manage, engineer, learn, improve, etc.) it. For example, to evaluate the maintenance process to improve it.
  - **Perspective:** Examine the (cost, effectiveness, correctness, defects, changes, product metrics, reliability, etc.) from the point of view of the (developer, manager, customer, corporate perspective, etc.) For example, to examine the cost from the point of view of the manager.
  - **Environment:** The environment consists of the following: process factors, people factors, problem factors, methods, tools, constraints, etc.

2. Derive a set of questions:

- The purpose of the questions is to measure the goal as completely as possible. Questions are classified as product-related or process-related and provide feedback from the quality perspective.
- Product-related questions define the product and give input for the evaluation of the product concerning a particular quality (e.g. reliability, user satisfaction).
- Process-related questions include the quality of use, domain of use, the effort of use, effect of use, and feedback from the user.

3. Develop a set of metrics:

- In this step, the actual data needed to answer the questions are identified and associated with each of the questions. As data items are identified, it must be understood how valid the data item will be for accuracy and how well it responds to the specific question.

- The metrics should be objective and subjective and should have interpretation guidelines, i.e. what value of the metric specifies the product's higher quality. Generally, a single metric will not answer a question, but a combination of metrics is needed. Metrics generally address the issues such as Size, people, time, cost, defects, difficulty (complexity), and communication issues. Fig. 4.11 illustrates how several metrics might be generated from a single goal.
- As shown in Fig. 4.11 the overall goal is to evaluate the effectiveness of the coding standard. That is, we want to know if code produced by following the standard is superior to code produced without following it.
- To decide the standard's effectiveness, several key questions are asked. First, it is important to know who is using standard, so that we can compare the productivity of the coders who use standard with the productivity of those who do not use standard. Likewise, we need to compare the quality of the code produced with the standard with the quality of non-standard code.
- Once these questions are identified, the questions are analyzed to determine what must be measured to answer these questions.
- For example, to understand who is using the standard, it is necessary to know what proportion of coders is using the standard.
- It is also important to have an experience profile of the coders, explaining how long they have worked with the standard, the environment, and the language. Productivity can be measured in terms of size and effort. Similarly, quality may be measured in terms of the number of errors found, etc.

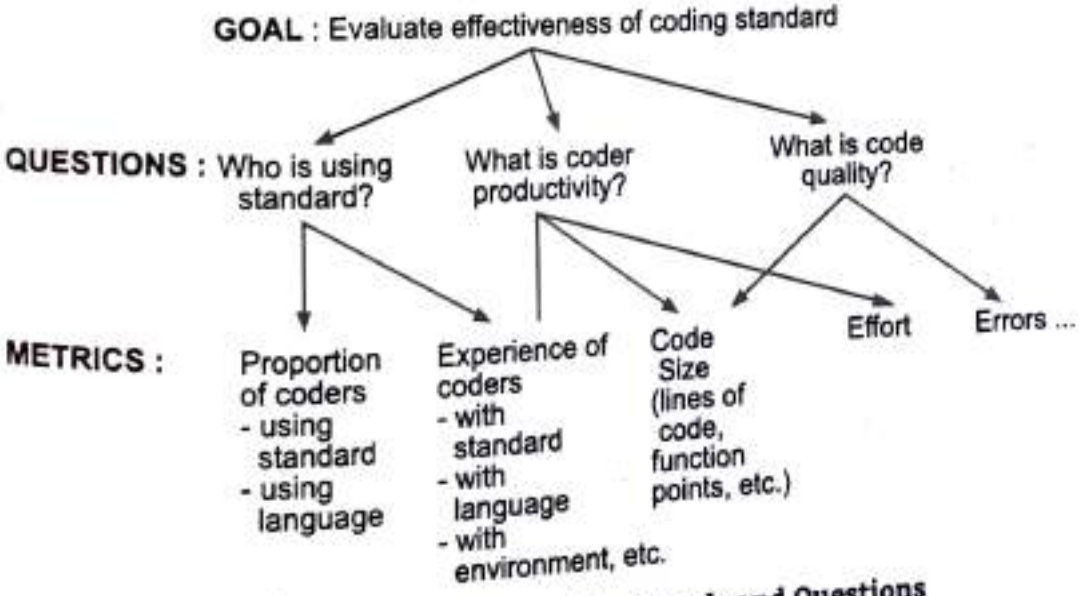


Fig. 4.11: Deriving Metrics from Goals and Questions

4.2.5 GQM Examples

Example 1: GQM tree helps to decide about when to distribute the software (based on the reliability of the software).

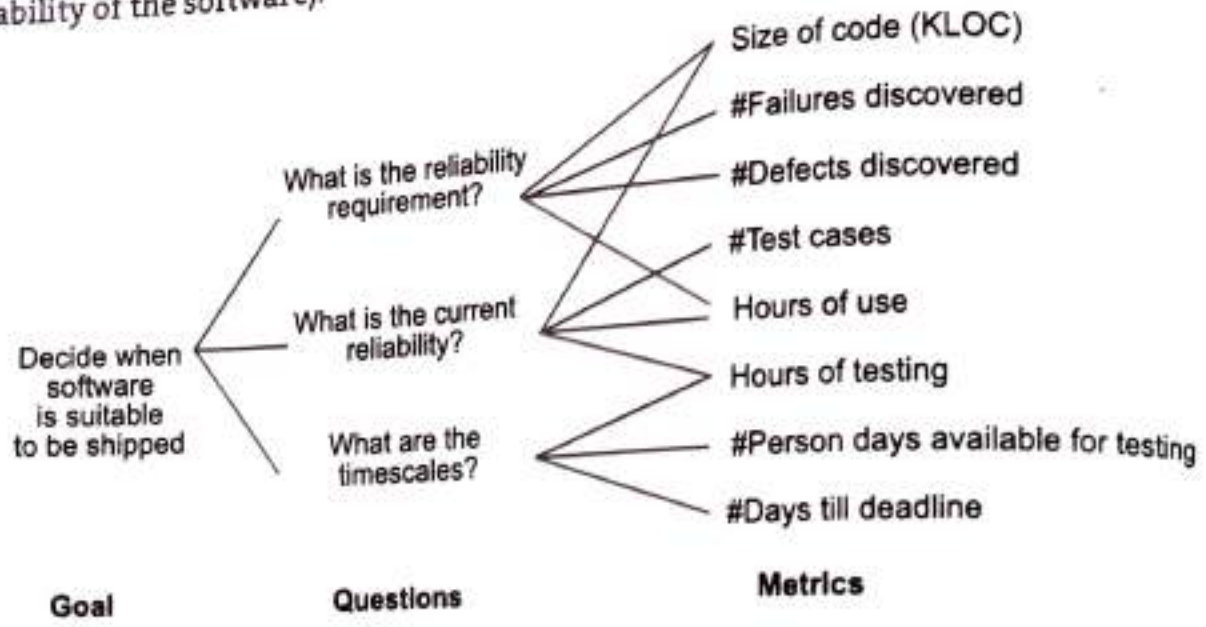


Fig. 4.12: GQM Tree on Software Reliability

Example 2: GQM tree on improving the maintainability of software.

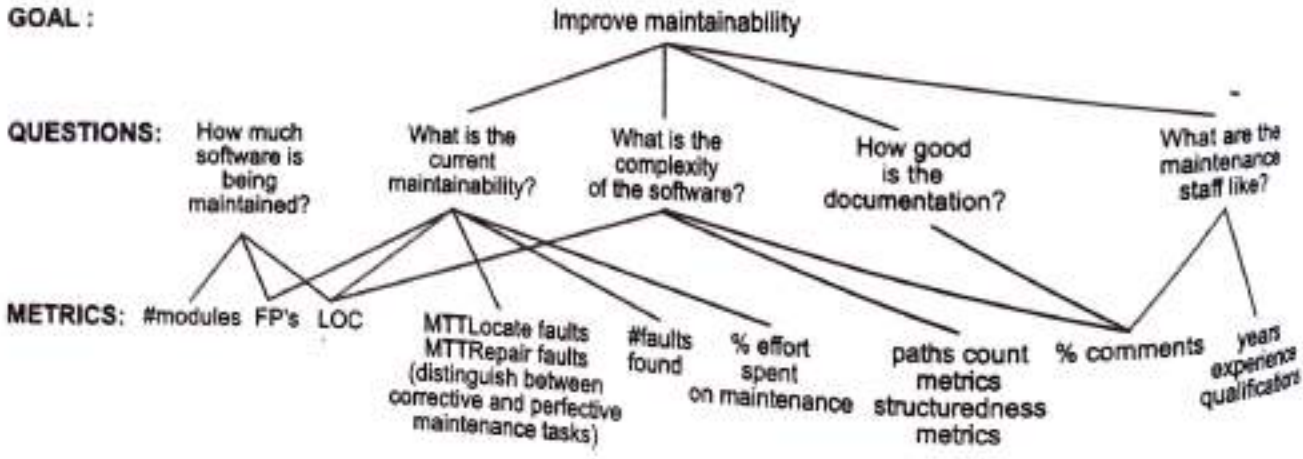


Fig. 4.13: GQM Tree on Improving Maintainability of Software

Example 3: GQM tree for the Inspection process.

- GQM method was used for identifying the most appropriate metrics for assessing the inspection process. Chosen metrics were primarily processed metrics.

Goal	Questions	Metrics
Plan	– How much does an inspection process cost? – How much calendar time does an inspection process take?	Average effort per KLOC. % of reinspections. Total KLOC inspected.
Monitor and Control	– What is the quality of inspected software? – To what degree did the staff conform? – What is the status of the inspection process?	Average faults detected per KLOC. Average inspection rate. Average preparation rate. Average LOC inspected. Total KLOC inspected.
Improve	– How effective is the inspection process? – What is the productivity of the inspection process?	Defect removal efficiency. Average faults detected per KLOC. Average inspection and preparation rate. Average LOC inspected.

Fig. 4.14: GQM tree for Inspection Process

Example 4: GQM tree for the productivity of the team.

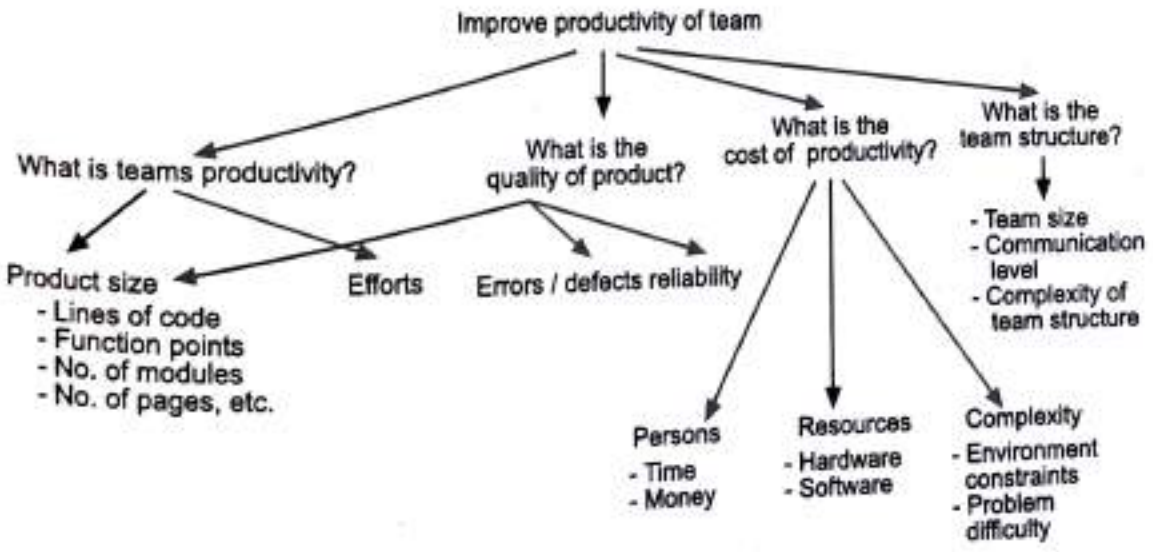


Fig. 4.15: Productivity of the Team

Example 5: Goal - Question-Metric approach for improving software development process.

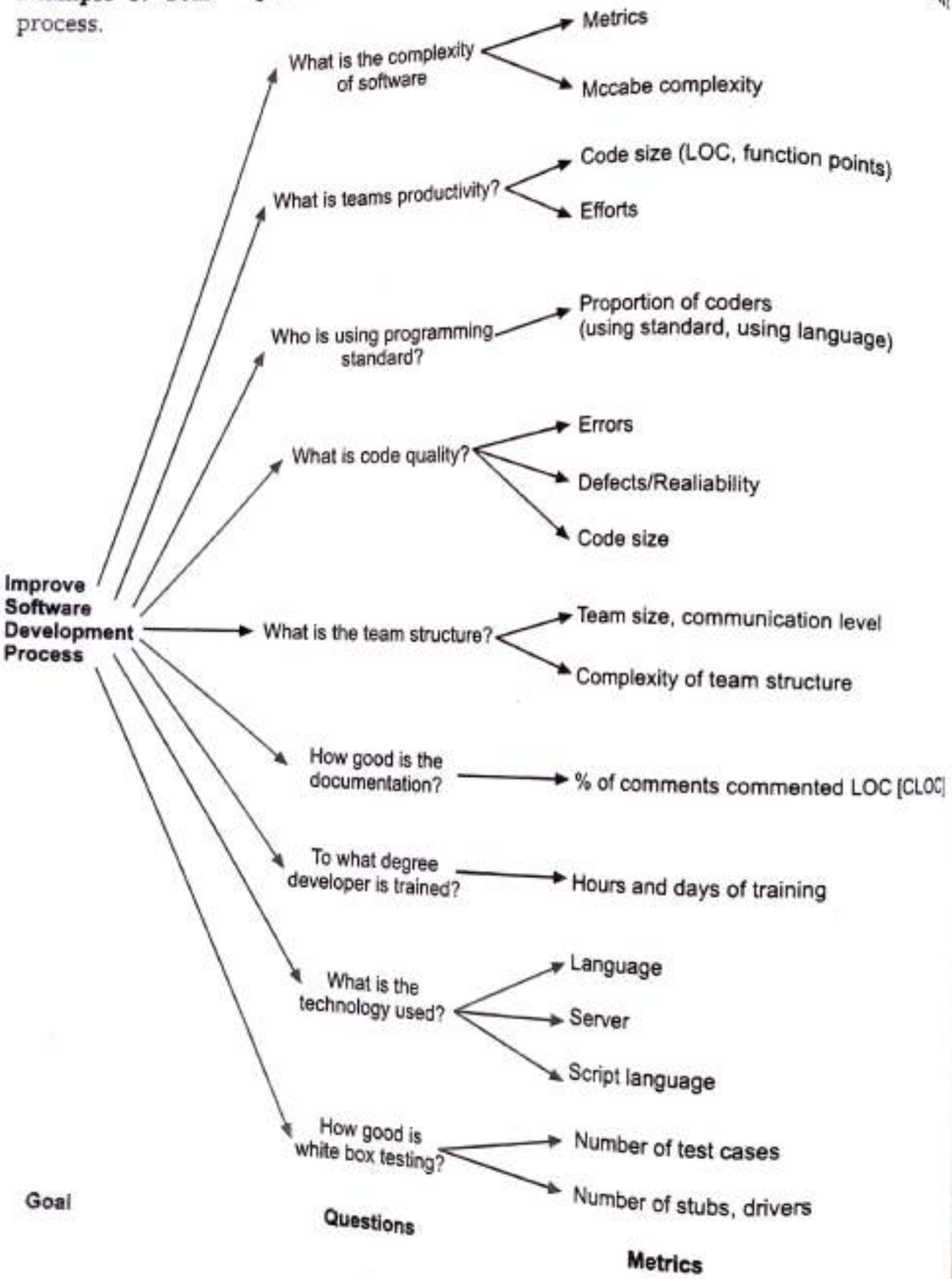


Fig. 4.16: GQM Approach

Example 6: GQM tree for achieving better software usability.

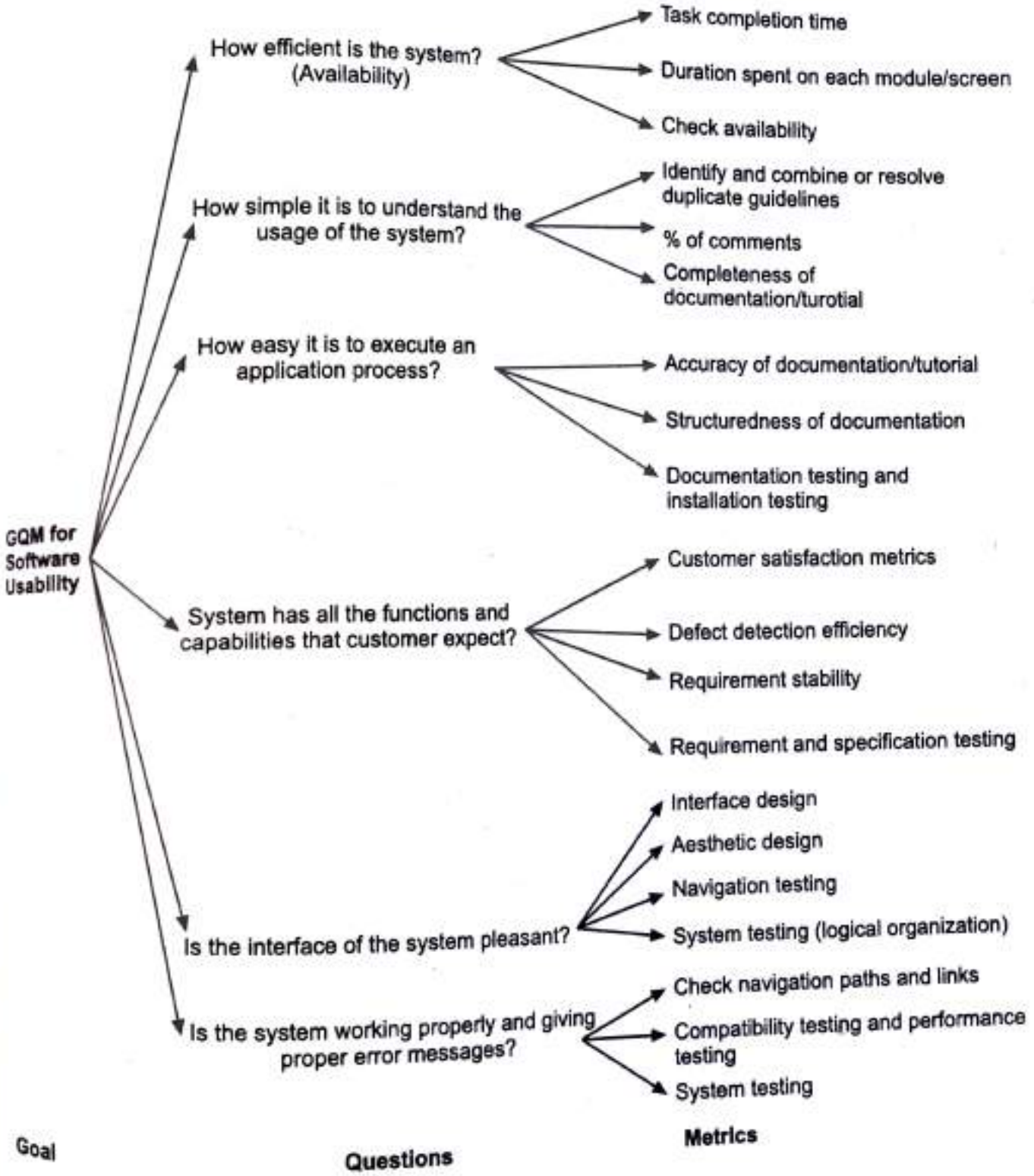


Fig. 4.17: GQM Tree

Example 7: GQM method for identifying software measures.

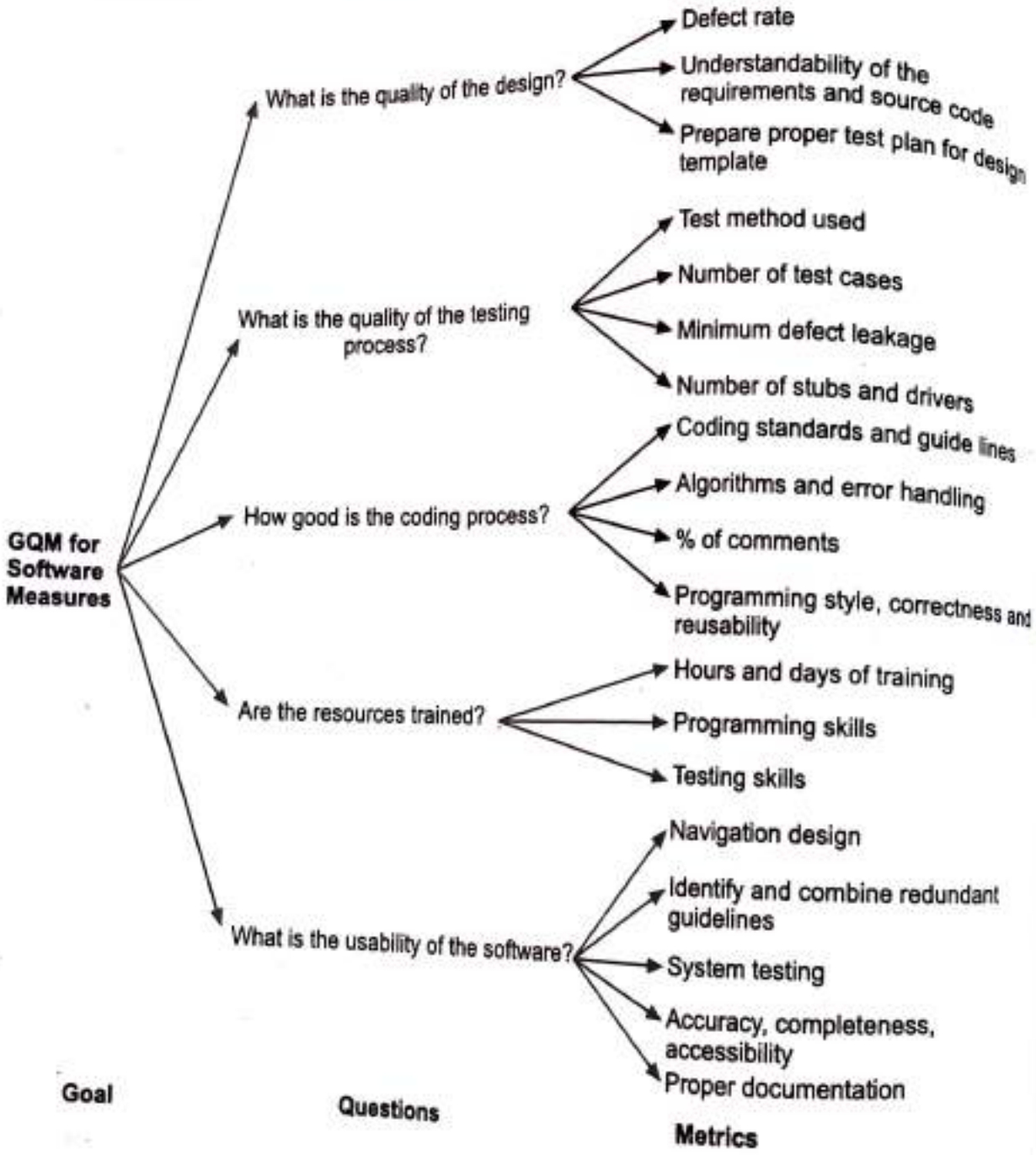


Fig. 4.18: GQM Method

- Some of the goals and measurement areas for Software Development are listed below :

Goal 1: Improve Project Planning

Question 1.1: What was the accuracy of estimating the actual value of the project schedule?

Metric 1.1: Schedule Estimation Accuracy (SEA)

$$\text{SEA} = \frac{\text{Actual project duration}}{\text{Estimated project duration}}$$

Question 1.2: What was the accuracy of estimating the actual value of project effort?

Metric 1.2: Effort Estimation Accuracy (EEA)

$$\text{EEA} = \frac{\text{Actual project effort}}{\text{Estimated project effort}}$$

Goal 2: Increase Software Reliability:

Question 2.1: What is the rate of software failures, and how does it change over time?

Metric 2.1: Failure Rate (FR)

$$\text{FR} = \frac{\text{Number of failures}}{\text{Execution time}}$$

- In the GQM model, the goal tells us why we are measuring, and the questions and metrics tell us what to measure.

4.2.6 Size Oriented Software Metrics

- Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced.
- The size could be in the form of page numbers, paragraph numbers, number of functional requirements, etc.
- Productivity is defined as KLOC/EFFORT, where effort is measured in person months.
- Size-oriented metrics depend on the programming language used.
- As productivity depends on KLOC, so assembly language code will have more productivity.
- LOC measure requires a level of detail that may not be practically achievable.
- There are a thousand lines of code (KLOC) that are often chosen as the normalization value.
- Metrics include:
  - Errors per KLOC - Errors per person-month.
  - Defects per KLOC - KLOC per person-month.
  - Dollars per KLOC - Dollars per page of documentation.
  - Pages of documentation per KLOC.
- This metric is not universally accepted as the best way to measure the software process.

4.2.7 Function - Oriented Metrics

- This metric uses a measure of the functionality delivered by the application as a normalization value.
- Function-oriented metrics were first proposed by Albrecht.

- The most widely used metric of this type is the function point:
  FP = count total * [0.65 + 0.01 * sum (value adj. factors)]
- Note that FPs try to quantify the functionality of the system, i.e., what the system performs. This is taken as the method of measurement as FPs cannot be measured directly. Also, note that FP is a collection of multiple software features/characteristics, not a single one.

1. FPs of an application are found out by counting the number and types of functions used in the applications. Various functions used in an application can be put under five types as shown in Table 4.2.

Table 4.2: Types of FP Attributes

Sr. No.	Measurement Parameter	Examples
1.	Number of external inputs (EI)	Input screen and tables.
2.	Number of external outputs (EO)	Output screens and reports.
3.	Number of external inquiries (EQ)	Prompts and interrupts.
4.	Number of internal files (ILF)	Database and directories.
5.	Number of external interfaces (EIF)	Shared databases and shared routines.

- Information domain values are defined in the following manner:
  - **Number of external inputs:** Each external input that provides distinct application-oriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.
  - **Number of external outputs:** Each external output that provides application-oriented information to the user is counted. In this context, output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.
  - **Number of external inquiries:** An inquiry is defined as an online input that results in the generation of some immediate software response in the form of online output. Each distinct inquiry is counted.
  - **Number of internal files:** Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.
  - **Number of external interfaces:** All machine-readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.

2. FP characterizes the complexity of the software system and hence can be used to depict the project time and the manpower requirement.
3. The effort required to develop the project depends on what the software does.
4. FP is a programming language independent.
5. FP method is used for data processing systems, business systems like information systems.

- The 5 parameters mentioned above are also known as **information domain characteristics**.

Example:

1. Computer the FP value for a project with the following information domain characteristics:

   No. of user inputs = 32

   No. of user outputs = 60

   No. of user inquiries = 24

   No. of user files = 08

   No. of External interfaces = 2

   Assume that all complexity adjustment values are average. Assume 14 algorithms have been counted.

Solution: Given information:

No. of user inputs = 32

No. of user outputs = 60

No. of user inquiries = 24

No. of user files = 08

No. of External interfaces = 2

And complexity adjustment values are average, so,

Measurement Parameter	Count	Weighting Factor
No. of user inputs	32 * 4 =	128
No. of use outputs	60 * 5 =	300
No. of user inquiries	24 * 4 =	96
No. of use files	08 * 10 =	80
No. of External interfaces	2 * 7 =	14
Count total		**618**

So, FP = Count total * [0.65 + 0.01 * Σ(Fi)]

$\Sigma(Fi) = 14 * 3 = 42$ (because in the question it is given that complexity adjustment values are average and all 14 algorithms are counted)

- $FP = 618 * [0.65 + 0.01 * 42]$
- $FP = 661$

4.3 CYCLOMATIC COMPLEXITY METRICS

- Code complexity testing is a method to determine how thoroughly a program should be tested. This is achieved by various complexity metrics, which are based on a combination of the size of the program and the complexity of the logic used to control the program.
- Complexity metrics can be used as an input for risk-based testing. High complexity would lead us to assume a higher likelihood of failure. By breaking down a system into programs we can use the program complexity as a component of the likelihood factor used in calculating risk levels.
- In defining the risk level, the tester still needs to assess the impact of the failures for each program, as well as the likelihood of the defect turning into a failure. Various complexity metrics are:
  1. Lines of Code.
  2. McCabe's Cyclomatic Complexity.
  3. Halstead's Software Science.

1. Lines of Code (LOC):

- Simple metric: Count the number of lines of code in the program and use count as a measure of complexity.

2. Cyclomatic Complexity: [S-22, 23; W-22]

- It is software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for Cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.
- Cyclomatic complexity has a foundation in graph theory and is computed in one of three ways:
  1. The **NUMBER OF REGIONS** corresponds to the Cyclomatic complexity.
  2. Cyclomatic complexity, V(G), for a flow graph G is defined as:

     V(G) = E − N + 2

     Where, E is the number of edges, and N is the number of nodes.

3. Cyclomatic complexity, V(G), for a flow graph G is also defined as:

   V(G) = P + 1

   Where P is the number of **PREDICATE NODES** contained in the flow graph G.

Example 1: Calculate Cyclomatic complexity for the given code in Fig 4.19(a). W-22; S-23

Solution:

- The Cyclomatic complexity can be computed using each of the formulas discussed above:
  - The flow graph has **three** regions.
  - V(G) = 8 edges – 7 nodes + 2 = **3**
  - V(G) = 2 predicate nodes + 1 = **3** (Predicate nodes – B,E)

```
A. input(score)
B. if score < 45
C.     then print ('fail')
D.     else print ('pass')
E. if score > 80
F.     then
print('distinction')
G. end
```

Fig. 4.19 (a) : Sample Code

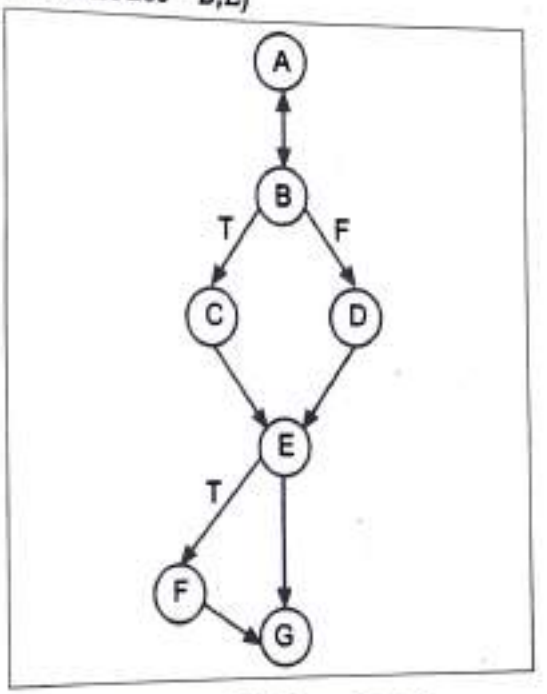


Fig. 4.19 (b) : Flow Graph

- Complexity is additive. If module A has a complexity of 7 and module B has a complexity of 9, then their combination is the complexity of 16.

Example 2: Calculate Cyclomatic complexity for the given code:

```
{ int i, j, k;
for (i=0 ; i<=n ; i++)
p[i] = 1;
for (i=2 ; i<=n ; i++)
{
   k = p[i]; j=1;
   while (a[p[j-1]] > a[k] {
   p[j] = p[j-1];
```

```
        j--;
    }
    P[j]=k;
    }
```

Solution:

- We draw the following control flow graph for the given code:

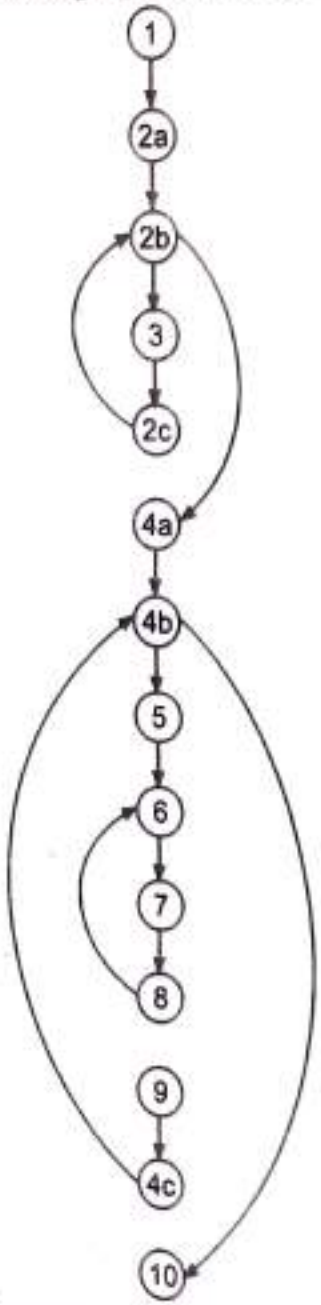


Fig. 4.20: Control Flow Graph

- Using the above control flow graph, the Cyclomatic complexity may be calculated as:

Method 1:

Cyclomatic Complexity

= Total number of closed regions in the control flow graph + 1

= 3 + 1

= 4

Method 2:

Cyclomatic Complexity

= E – N + 2

= 16 – 14 + 2

= 4

Method 3:

Cyclomatic Complexity

= P + 1

= 3 + 1

= 4

- For small programs, Cyclomatic complexity can be calculated manually but automated tools are essential for big and complex programs. Based on the complexity number, one can conclude what actions need to be taken for the complexity measure using Table 4.3.

Table 4.3: Meaning of Complexity Measure

Complexity	Meaning
1 – 10	Well-written code, testability is high; cost/effort to maintain is low.
10 – 20	Moderately complex, testability is medium; cost/effort to maintain is medium.
20 – 40	Very complex, testability is low; cost/effort to maintain is high.
> 40	Difficult and complex and tedious to test.

3. **Halstead's Software Science:**

- Halstead's metric is based on the use of operators (for example, keywords) and operands (e.g. variable names, database objects) used in a program.
- Primitive measures of Halstead's software science are:

  n_1 = Number of distinct operators in program.

  n_2 = Number of distinct operands in program.

  N_1 = Number of operator occurrences.

  N_2 = Number of operand occurrences.

- Based on these primitive measures, Halstead developed a system of equations expressing the total vocabulary, the overall program length, the potential minimum volume for an algorithm, the actual volume (number of bits required to specify a program), the program level (a measure of software complexity), program difficulty, and other features such as development effort and the projected number of faults in the software.

- Halstead's major equations include the following :

Vocabulary (n)	$n = n_1 + n_2$
Length (N)	$N = N_1 + N_2$ $= n_1 \log_2(n_1) + n_2 \log_2(n_2)$
Volume (V)	$V = N \log_2 (n)$ $= N \log_2 (n_1 + n_2)$
Level (L)	$L = V^*/V$ $= (2/n_1)\ (n_2/N_2)$
Difficulty (D) (Inverse of L)	$D = V/V^*$
Effort (E)	$E = V/L$
Faults (B) (Bug prediction)	$B = V/S^*$ $= (N_1 + N_2) \log_2\ (n_1 + n_2)/3000$

S* is the mean number of decisions between errors (S* is 3,000 according to Halstead).

Deriving Test Cases:

- The basis path testing method can be applied to a detailed procedural design or source code. Basis path testing can be seen as a set of steps.
  - Using the design or code as the basis, draw an appropriate flow graph.
  - Determine the Cyclomatic complexity of the resultant flow graph.
  - Determine a basis set of linearly independent paths.
  - Prepare test cases that will force the execution of each path in the basis set.
- Data should be selected so that conditions at the predicate nodes are tested. Each test case is executed and contrasted with the expected result. Once all test cases have been completed, the tester can ensure that all statements in the program are executed at least once.

Software Quality Indicator:

- An indicator is a device or variable, which can be set to a prescribed state, based on the results of a process or the occurrence of a specified condition.
- A Software Quality Indicator is used to calculate and to provide an indication of the quality of the system by assessing system characteristics. The software quality indicators address management concerns. It is also used for decision-making by decision-makers. It includes a measure of the reliability of the code.
- An indicator usually compares a metric with a baseline or expected result.
- It acts as a set of tools to improve the management capabilities of personnel responsible for monitoring.

Software quality indicators extract from requirements are flexibility and adaptability.

Summary

- A software metric is a standard measure of the degree to which a software system or process possesses some property.
- Cyclomatic complexity is a software metric. It provides a quantitative measure of the logical complexity of a program. Cyclomatic complexity provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

Check Your Understanding

1. Number of errors found per person-hours expended is an example of a _____.
   (a) measurement (b) measure
   (c) metric (d) All of the mentioned
2. Function Points in software engineering was first proposed by_____.
   (a) Boehm (b) Jacobson
   (c) Albrecht (d) Babbage
3. Function Point Computation is given by the formula_____.
   (a) FP = count total * [0.65 + 0.01] * sum(Fi)
   (b) FP = count total * [0.65 + 0.01 * sum(Fi)]
   (c) FP = [count total * 0.65] + 0.01 * sum(Fi)
   (d) FP = [count total * 0.65 + 0.01] * sum(Fi)
4. Which of the following is an indirect measure of the product?
   (a) Quality (b) Complexity
   (c) Reliability (d) All of the Mentioned
5. The task of project indicators is _____.
   (a) of ongoing projects help in the valuation of the status.
   (b) potential risk tracker.
   (c) track potential risk and help in the assessment of the status of an ongoing project.
   (d) None of these.
6. Metrics are developed based on_____ in size-oriented metrics.
   (a) number of Functions (b) number of user inputs
   (c) number of lines of code (d) amount of memory usage

7. Cyclomatic complexity of a flow graph G with N vertices and E edges is
   (a) V(G)=E+N-2 (b) V(G)=E-N+2
   (c) V(G)=E+N+2 (d) V(G)=E-N-2
8. Which of the following is a software metric that provides a quantitative measure of the logical complexity of a program?
   (a) Cyclomatic Complexity (b) LOC
   (c) Function Point (d) Index matrix
9. Which of the following is not a direct measure of the SE process?
   (a) Efficiency (b) Cost
   (c) Effort Applied (d) All of the mentioned
10. Which of the following are advantages of using LOC (lines of code) as a size-oriented metric?
   (a) LOC is easily computed.
   (b) LOC is a language-independent measure.
   (c) LOC can be computed before a design is completed
   (d) LOC is a language-dependent measure.

Answers

1. (c)	2. (c)	3. (b)	4. (d)	5. (c)
6. (c)	7. (b)	8. (a)	9. (a)	10. (a)

Practice Questions

Q.I Answer the following questions in short.

1. What is a Software Metrics?
2. Which are the types of metrics?
3. What are direct and indirect measures?
4. What is Halstead's metric?
5. Which are the properties of metrics?

Q.II Answer the following questions.

1. Explain in short a concept of Complexity Metrics.
2. Explain the GQM model of Metrics.
3. Describe the capability maturity model.
4. Describe size-oriented metrics.
5. Explain function-oriented metrics.

Q.III Define the terms.

1. Indicator
2. Software metrics
3. Lines of code
4. Function point
5. Product metrics

★★★

5... Testing for Specialized Environments

Learning Objectives ...

- To learn how to test graphical user interfaces.
- To study testing client/server Architectures.
- To study testing documentation and help facilities.
- To understand real-time system testing.

5.1 INTRODUCTION

- The necessity for a specialized testing technique has risen as computer software has become more complicated. The White Box and Black Box testing methods are applicable across all environments, architectures, and applications, but unique guidelines and approaches to testing are sometimes warranted.

5.2 TESTING GUI'S [S-22, 23; W-22]

- Graphical User Interfaces (GUIs) present interesting challenges for software engineers. Because of reusable components provided as part of GUI development environments, the creation of the user interface has become less time-consuming and more precise. But, at the same time, the complexity of GUIs has grown, leading to more difficulty in the design and execution of test cases.
- Because many modern GUIs have the same look and feel, a series of standard tests can be derived. Finite-state modeling graphs may be used to derive a series of tests that address specific data and program objects that are relevant to the GUI.
- Due to the large number of permutations associated with GUI operations, testing should be approached using automated tools. A wide array of GUI testing tools has appeared on the market over the past few years.

- This is the most important part of the application along with functionality, as it may affect usability perspective. The Graphical User Interface, or GUI, is the method through which you interact with a software application.
- GUI testing involves determining how the program responds to keyboard and mouse events, as well as how different GUI components such as menu bars, toolbars, dialogues, buttons, edit fields, list controls, graphics, and so on react to user input.

5.2.1 Characteristics of GUI

- Graphical user testing is also known as GUI testing or UI testing.
- Here's a list of seven important traits common to a good user interface.

1. **Follows Standards and Guidelines:**

- The single most important user interface trait is that your software follows existing standards and guidelines.
- Everything is defined from when to use checkboxes instead of an option button when to use the information, warning, and critical messages as shown in Fig. 5.1.

Fig. 5.1: Testing of GUI

2. **Intuitive:**

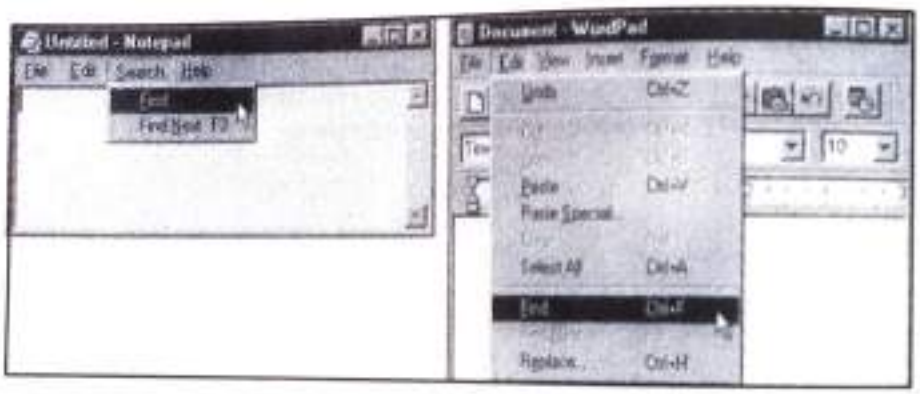
- When you are testing a user interface, consider the following things and how they might apply to judge how intuitive your software is:
  - Is the user interface simple and clutter-free? The services you want or the responses you need should be evident and available when you require them.
  - Is the UI organized and laid out well? Does it allow you to easily get from one function to another? Is what to do next obvious? At any point can you decide to do

nothing or even back up or back out? Does the menu or window give too many details?

- Does it contain excessive functionality? Do too many features complicate your work? Do you feel like getting information overload?

3. **Consistent:**

- Consistency within your software and with other software is a key attribute. The user expects that if they do something in a certain way in one program, another will do the same operation in the same way.
- For instance, In Notepad, Find is accessed through the Search menu or by pressing F3. In WordPad, a very similar program is accessed through the Edit menu or by pressing Ctrl+F. Such inconsistencies frustrate users as they move from one program to another.



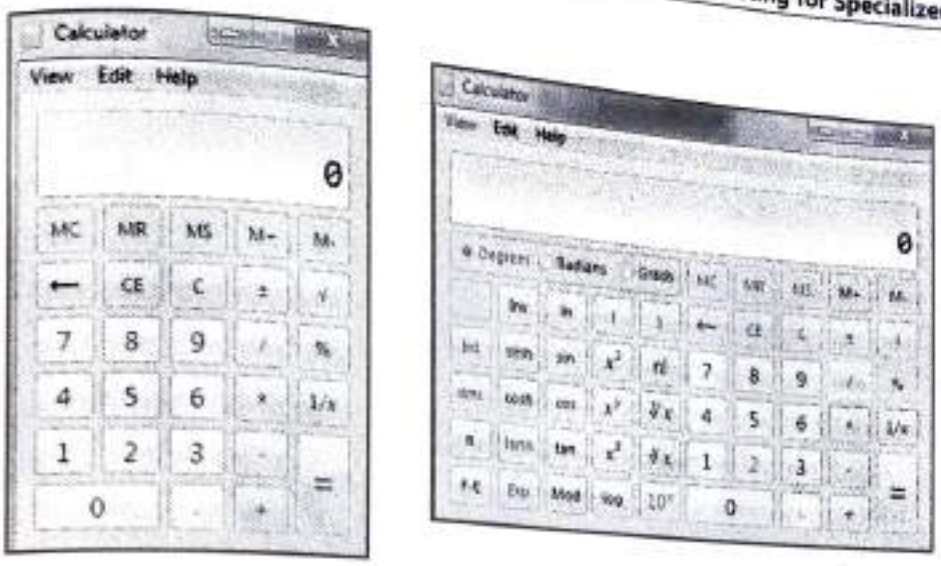
(a) (b)

Fig. 5.2: Shortcut keys and Menu selection

- In Windows, pressing F1 you should get help.
- **Terminology and Naming:** Are the same terms used throughout the software? Are features named consistently? For example, is Find always called 'Find', or is it sometimes called 'Search'?
- **Placement for buttons such as OK and Cancel:** In Windows, OK is always on the top or left and Cancel on the right or bottom? The Mac OS places OK on the right side of the screen. Keyboard equivalents to onscreen buttons should also be consistent. For example, the Esc key always does a cancel, and Enter does an OK.

4. **Flexible:**

- Users want to choose what they want to do and how they want to do it rather than having too many options.
- The Windows Calculator as shown in Fig. 5.3 has two views: Standard and Scientific. Users can decide which one they need for their task or the one they're most comfortable using.



(a) (b)

Fig. 5.3: Calculator Window

5. **Comfortable:**

- Software should be comfortable to use. It should not make it difficult for a user to do the work.

(i) **Appropriateness:** Software should appear and feel appropriate for the task at hand and the audience for whom it was created. Many colors and strong sound effects are usually not appropriate for a financial business application. The rules of a space game, on the other hand, will be far more flexible. For the task given, software should be neither too large nor too simple.

(ii) **Error handling:** A program should warn users before a critical operation and allow users to restore data lost because of a mistake.

(iii) **Performance:** Being fast is not always a good thing. For example, the status bars show how much of the work has been completed and how much is left to go.

6. **Correctness:**

- When testing for correctness, test whether the GUI does what it's supposed to do.
- Check to see if the GUI is displaying what you want.
- Is the document that appears on the screen; the same as what is saved to the disc when you click the Save button?
- Does the document match properly to the original when you reload it?
- Is the result the same as what you see on the screen when you print it?

7. **Useful:**

- The final trait of a good user interface is whether it is useful. Check that the features specified in the software must be useful.

5.2.2 Advantages of GUI Testing

1. Consistency of screen layouts and designs improves the usability of an application.
2. Tab sequence provides a logical way of doing the sequence.
3. Good GUI improves the look and feel of the application. It helps in the psychological acceptance of the application by the user.

5.2.3 Types of GUI

- GUI testing can be decomposed into two types:
  1. Windows Compliance Testing.
  2. Web app Interface Testing.

A. Windows Compliance Testing

- Windows compliance testing is done for:
  1. Application.
  2. Each window in the Application.
  3. Text boxes, Option (Radio) buttons, Checkboxes, Command buttons, Drop-down List boxes, Combo boxes, and List boxes.

1. Application:

- Double-click the application's icon to launch it:
  - The loading message should show the application name, version number, and a bigger pictorial representation of the icon.
  - The caption for the application's main window should match the caption for the icon in Program Manager.
  - When you close the application, you should get a "Are you sure?" dialogue box.
  - A Help button should be included on every screen; F1 should be used to provide on-screen help to the user.

2. Each window in the application:

- Check all text on the window for Spelling/Tense and Grammar.
- Use TAB to move focus around the Window. Use SHIFT+TAB to move focus backward.
- Tabbing to an entry field with text in it should highlight the entire text in the field.
- A field should not be focused if it is disabled (greyed). Users should not be able to select them using the mouse or the TAB key.
- All text should be left-justified, followed by a colon right to it.
- All tab buttons should have a distinct letter.

3. **Text boxes, Option (Radio buttons), Checkboxes, Command buttons, Drop-down list boxes, Combo boxes, and List boxes:**

- For Text Boxes, SHIFT and Arrow should select characters. Selection should also be possible with a mouse. Double click should select all text in a box.
- For Checkboxes, clicking with the mouse on the box, or on the text should SET/UNSET the box. SPACE should do the same.
- For Command Buttons, all buttons except for OK and Cancel should have a letter Access to them. If there is a Cancel Button on the screen, then pressing <ESC> should activate it.
- For Drop-down List Boxes, pressing a letter should bring you to the first item in the list which starts with that letter. Pressing 'Ctrl+F4' should open/drop down the list box.

B. **Web app Interface Testing**

- In Web app interface testing, GUI testing exercises interaction mechanisms and validates aesthetic (graphical) aspects of the user interface.
- Web app interface testing consists of Interface testing strategy and Testing interface mechanisms.

1. **Interface Testing Strategy:**

- The overall strategy for interface testing is to uncover errors related to specific interface mechanisms. (For example Errors in the proper execution of a menu link or the way data is entered in a form etc.).

Objectives:

- To accomplish this strategy, several objectives must be achieved:
  - Interface features are checked for errors to guarantee that design principles, aesthetics, and associated visual content are available to the user. Features include font type, the use of color, frames, images, borders, tables, and related elements that are generated as Web app execution proceeds.
  - Individual interface mechanisms are tested in a manner that is analogous to unit testing. For example, Tests are designed to exercise all forms, client-side scripting, dynamic HTML, pop-up windows, etc.
  - The complete interface is tested against selected use-cases and navigation properties to uncover errors in the semantics of the interface. This testing approach is analogous to validation testing.
  - The interface is tested with a variety of environments (For example Browsers) to ensure that it will be compatible. These tests can also be considered as a part of configuration testing.

2. Testing Interface Mechanisms:

- When a user interacts with a Web app, the interaction occurs through one or more interface mechanisms. Testing considerations for each interface mechanism are:

(i) **Links:** Each navigation link is tested to ensure that the proper content object or function is reached. The Web engineer builds a list of all links associated with the interface layout (For example Menu bars, Index items) and then executes each individually. In addition, links within each content object must be exercised to uncover bad URLs or links to improper web pages or functions. Finally, links to external Web apps should be tested for accuracy and consistency.

(ii) **Forms:** Tests are performed to ensure that:

1. Label correctly identifies fields within the form and that mandatory fields are easily identified by the user.
2. Form fields have proper width and data types.
3. The server receives all information contained within the form and that no data are lost in the transmission between client and server.
4. Appropriate default values are used when the user does not select from a pull-down menu or set of buttons.
5. Browser functions (For example The "back" arrow) do not affect data entered in a form.
6. Error-checking scripts for data entered operate correctly and produce relevant error messages.

(iii) **Client-side scripting:** Black box tests are executed to uncover any errors whenever the script (For example JavaScript) is executed. These tests are often coupled with form testing because script input is derived from data provided by the form. A compatibility test should be conducted to ensure that the scripting language that has been chosen will work properly in the environment that supports the Web app.

(iv) **Dynamic HTML:** Each web page that contains dynamic HTML is executed to ensure that the dynamic display is correct. In addition, a compatibility test should be conducted to ensure that the dynamic HTML works properly in the configuration environment that supports the Web app.

(v) **Pop-up Windows:** A series of tests ensure that:

1. The pop-up is properly sized and positioned.
2. The pop-up does not cover the original Web app window.
3. The aesthetic design of the pop-up is consistent with the aesthetic design of the interface.

(vi) **Cookies:** Both server-side and client-side testing are required. On the server-side, tests should ensure that a cookie is properly constructed (contains correct data) and properly transmitted to the client-side when specific content or functionality is requested. In addition, the cookie is tested to ensure that its expiration date is correct. On the client-side, tests determine whether the Web app properly attaches existing cookies to a specific request, which is sent to the server.

(vii) **Application-Specific Interface Mechanisms:** Tests conform to a checklist of functionality and features that are defined by the interface mechanism.

- **For example,** the following is the checklist for shopping cart functionality defined for an e-commerce application:

1. Boundary test for the minimum and the maximum number of items that can be placed in the shopping cart.
2. Test a "check out" request for an empty shopping cart.
3. Test proper deletion of an item from the shopping cart.
4. Test to determine whether a purchase empties the cart of its contents.
5. Test to determine whether the Web app can retrieve shopping cart contents in the future (assuming that no purchase was made) if the user requests that contents be saved.

(viii) **GUI Checklist:**

- A complete review of the interface design model should be done, using a GUI checklist to ensure that every item has been exercised at least once.
- **For example,** Screen validation checklist.

Aesthetic Conditions:

1. Is the general screen background color correct?
2. Are all the screen prompts specified in the correct font?
3. Is the text in all fields specified in the correct font?
4. Are all the fields prompts aligned perfectly on the screen?
5. Is the screen resizable?
6. Is the screen minimizable?
7. Are all the field prompts spelled correctly?
8. Is all the error message text spelled correctly on this screen?
9. Assure that all windows and dialog boxes have a consistent look and feel.

Validation Conditions:

1. Does a failure of validation on every field cause a sensible user error message?
2. Is validation consistently applied at screen level unless specifically required at field level?

3. For all numeric fields, check whether negative numbers can and should be entered.
4. For all numeric fields check the minimum and maximum values along with some mid-range values are allowed?
5. Do all mandatory fields require user input?

Navigation Conditions:

1. Can the screen be accessed correctly from the menu?
2. Can the screen be accessed correctly from the toolbar?
3. Can the screen be accessed correctly by double-clicking on a list control on the previous screen?
4. Is the screen secured? i.e. Is the user prevented from accessing other functions where this screen is active and is this correct?

5.3 TESTING OF CLIENT/SERVER ARCHITECTURES

- Client/Server architectures as shown in figure 5.4, allow complex systems to be assembled from components. However, multiple operating systems, changing technologies, and greater architectural complexity make integration more difficult. Risks such as poor reliability, performance, configuration management, security, and other non-functional issues are more prominent. None of the risks in client/server are new, but there has been a change in emphasis. Since its purpose is to address risk, the emphasis of testing in Client/Server must change.
- The complexity of the client/server also makes testing more difficult. Complex systems can be delivered faster because they are assembled from bought-in components. Client/Server(C/S) architectures represent a significant challenge for software testers.
- Testing is often estimated to be in proportion with the development cost but in a typical Client/Server system, the development cost may be small compared with the overall cost of the system.
- In the Client/Server application, you have two different components to test. Application is loaded on server machine while the application executable component on every client machine. You will test broadly in categories like GUI on both sides, functionality, Load, Client/Server interaction, backend. This environment is mostly used in Intranet networks. You should be aware of the number of clients and servers and their locations in the test scenario.

Client/Server Testing Process:

- A Client/Server test strategy must identify the risks of concern and define a test process that ensures these risks are addressed. A problem is cheaper to fix if identified early. The test process should be aligned very closely to the development

process. Testing of a deliverable should occur as soon as possible after it has been built.

- **Functionality** is usually tested in three stages: a unit or component test, often performed by programmers; a system test, of the complete system in a controlled environment performed by a dedicated test resource; finally, an acceptance test under close to production conditions, often performed by users. The objectives, techniques, and responsibilities for these three test stages are directly comparable to test stages in more traditional host-based systems. The changed emphasis in testing Client/Server is associated with integration and non-functional testing.
- **Integration** is a big issue because Client/Server systems are usually assembled from around twelve components (for a simple 2-Tier system) to perhaps twenty components for complex architecture. These components are usually sourced from multiple suppliers. Although standards are emerging, Client/Server architectures often use components that have never been used before in combination.
- The total number of interfaces involved makes interface problems and inter-component conflicts more likely. Assumptions made by developers of one component from one supplier may contradict assumptions made by another developer. These problems may be faced for the first time in the installation. In Client/Server process, the system integrator takes responsibility for integration testing.
- **Performance** consistently presents a problem in Client/Server. 'Intelligent clients' may call and process large volumes of data across networks. The amount of code exercised in large numbers of architectural layers may be substantial. In general delay between distributed processes maybe only ten or twenty milliseconds with limited transactions. But when a transaction requires hundreds of network messages, then delay could be considerable.
- Other non-functional issues such as *security, backup and recovery,* and *system administration* are at high risk. What was taken for granted in a mainframe environment often creates problems in the Client/Server.
- The test strategy must address all these risks, but testing never happens 'at the end'. Testing occurs at all stages and includes review, walkthrough, and inspection. Developers should be responsible for the product they deliver and should test their code. System tests should cover non-functional areas as well as functionality.

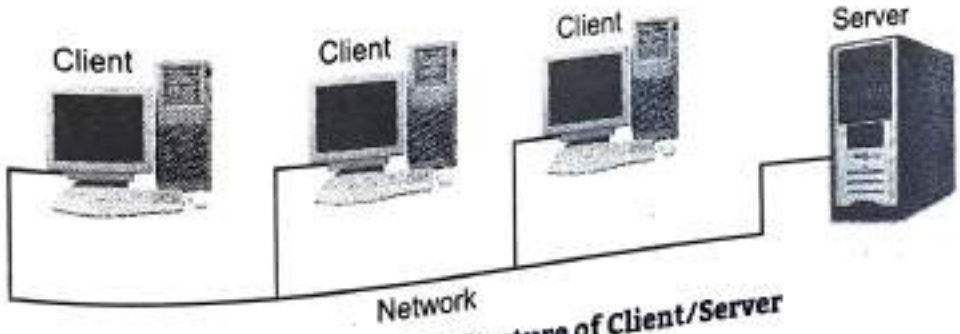


Fig. 5.4: Architecture of Client/Server

- Client/Server architecture testing involves component testing and integration testing followed by various specialized testing methods.
- Following testings are needed for the Client/Server system:
  1. **Component Testing:** For testing the client and server separately, the strategy and test plan must be defined. Simulators may need to be created to substitute related components when testing the targeted component.
     When testing a server, a client simulator may be required, and when testing a client, a server simulator may be required. We may have to test the network utilizing client and server simulators at the same time.
  2. **Integration Testing:** After successful testing of server, client, and network, they are integrated to form the system, and the system test cases are executed. Communication between client and server is tested in integration testing.

Special Testing in Client/Server:

- There are several special testing involved in the Client/Server application. Some of them are given below:
  1. **Performance Testing:** When several clients communicate with the server at the same time, the system's performance is evaluated. Client/Server applications may also be tested using volume testing and stress testing. We can test the system under maximum load as well as regular load as the number of customers is already known to the system. Stress testing may be done with a variety of user interactions.
     (** For more information about Load and Stress testing, refer to section 3.5.)
  2. **Concurrency Testing:** Client/Server design requires extensive testing. Concurrency testing is essential to understand how a system behaves in situations where multiple users may be accessing the same data at the same time.
  3. **Disaster Recovery/Business Continuity Testing:** When a client and a server communicate with one other, the communication may get interrupted for a variety of reasons, such as the failure of either the client or the server, or the failure of the connection that connects them. To understand how the system responds to such disasters, disaster recovery and business continuity tests may be included.
  4. **Testing for extended periods:** In Client/Server applications, the server is rarely shut down unless there are certain agreed-upon Service Level Agreements (SLAs) that allow the server to be taken down for maintenance. The server is running 24 hours a day, seven days a week for an extended length of time. To determine if the network and server's service quality deteriorates over time owing to factors such as memory leakage, testing must be conducted over a long period.
  5. **Compatibility Testing:** When users utilize the client and server in production, they may be placed in distinct environments. Servers can run on a variety of hardware, software, and operating systems. Clients may have large differences

from predicted environmental factors. Testing must guarantee that performance is maintained across a wide range of hardware and software combinations, and that the user's work is unaffected by mismatched setups. Similarly, any limitations must be communicated to the potential user.

5.3.1 Web-Based Testing

- A web-based application may be accessed and utilized across a network, such as the Internet, Intranet, or Extranet. The Internet is a global network of interconnected networks. Intranet is the network that internal employees use within the organization, while extranet is the network that the organization uses internally as well as with its business partners.
- Web-based architecture is an extension of Client/Server architecture. There is a difference between both architectures. In Client/Server architecture the client workstation has the application software which is used to communicate to the application server but in the web-based application the client machine has the web browser and these client machines are networked to the web server by either LAN (Local Area Network) or WAN (Wide Area Network).
- The testing is based on your web testing requirements but the following is the standard checklist of web testing:

1. **Functionality Testing:**

- The term "functionality testing" (also known as "component testing") refers to the process of determining whether or not each component functions as planned. Its goal is to ensure that the functional specifications in the functional documents are correct. Its goal is to assess if the software application satisfies the expectations of the user. It compares the behavior of a software program to the test requirements.

The following activities are included in this testing:

(i) **Link Testing:** Check different links on web pages:

- Internal links
- Outgoing links
- Mailto Links
- Anchor Links

(ii) **Web Form Testing:** The main purpose of Form is to get the information from the user and store it in the database. Below are some test cases which should be considered while doing form testing:

- The first thing to test in form is the Validation of each form field. Here two types of validations need to be considered – "Client-side" and "Server-side" validations.
- Examine all Mandatory fields. Check to see whether a user hasn't filled out the necessary field, and display a mandatory error message if they haven't.
- Add information using Form and update information using Form.

- Tab orders.
- Check for the default values of fields.
- In Negative testing, enter invalid inputs to the form fields.

(iii) **Cookies Testing:** Cookies are little text files that are encrypted and kept in browser folders. Browsers utilize these files to access cookies. Cookies are used to keep login sessions active. In general, cookies include user information that is used to communicate across websites. Information is stored in cookies (similar to Sessions) and may be accessed for websites. Cookies can be enabled or disabled in browser settings. Session cookies and persistent cookies are the two sorts of cookies. Session cookies are active as long as the browser associated with that cookie is open. Session cookies are immediately removed when the relevant browser is closed. Persistent cookies are stored on the user's computer indefinitely. In Cookies testing, the following should be tested:

- Test the application by disabling the Cookies.
- Test the application after corrupting the cookies.
- Test the application's functionality after erasing all cookies from the website you're testing.
- Check whether website cookies are working or not on different browsers.
- Check if cookies for authenticated login are working or not.
- Check the behavior of the application after deleting the cookies (sessions) by clearing the cache or after cookies expired.
- Check the behavior of the application after deleting the login cookies (sessions).

2. **Usability Testing:** This would include:

(i) **Navigation Testing of the Website:**

- All possible options like Menus, Links, or buttons on web pages should be visible and accessible from all the web pages.
- Web page navigation should be easy to use.
- Help instructions should be clear and should satisfy the purpose.
- All options on the header, footer, and left/right navigation should be consistent throughout the pages.

(ii) **Content Testing of the Website:**

- No spelling or grammatical errors/mistakes in content throughout the page.
- The desired text should be present with Images.
- No broken images.
- Follow certain web page content creation guidelines.
- All content should be legible and easy to understand.
- Dark color infuriates users, so avoid using dark colors in the theme.

- Proper size images should be placed on the web page.
- All the anchor text links should be working properly.

Compatibility Testing:

Compatibility testing is the non-functional aspect of software testing. It ensures that the program works in the supported environments. The customer uses online applications on various operating systems, browsers, hardware platform computes power, databases, and networking hardware bandwidth handling capability. The compatibility testing is to make sure that "Is web application work correctly across different devices?"

This includes the following tests:

(i) **Browser Compatibility Test:** Web apps operate differently in various browsers. The goal of browser compatibility testing is to guarantee that consumers have no problems viewing the site using various browsers. Browser compatibility testing ensures that your web application is shown correctly across a variety of browsers. The tests should also ensure that AJAX, JavaScript, and authentication are all working properly.

(ii) **OS Compatibility:** With new technologies, complex graphical designs and APIs are used, which may not operate on all operating systems. Objects such as text boxes and buttons may also show differently on various operating systems. As a result, web application testing should be performed on many operating systems such as Windows, MAC, Solaris, UNIX, and Linux with various OS flavors.

(iii) **Mobile Browsing:** In the latest Mobile technology tester should test Mobile Browser Compatibility too.

Database Testing:

Data reliability is a key part of Database testing.

Testing activities would include:

(i) Check if queries are executed without any errors.

(ii) Creating, updating, or deleting data in the database should maintain data integrity.

(iii) More time should not take to execute the queries, if required tune the queries for better performance.

(iv) Check load on the database while executing heavier queries and check the result.

Interface Testing:

Interface testing should cover mainly three areas - Web Server, Application Server, and Database Server.

Ensure that the communications between these servers are carried out properly. Check if any servers' connections have been reset or lost. Check if any requests are interrupted in the middle and how the program responds.

- The tester should check whether the errors (if any) received from a web server or database server to the application server are handled properly and displayed to the user.
  (i) **Web Server:** Check if all web requests are processed.
  (ii) **Application Server:** Check if the request is being sent to the server successfully and is being displayed correctly. Check whether errors are being noticed and shown to the admin user.
  (iii) **Database Server:** Check if the database server returns the correct result on the query request.
  Check if all three servers are connected and the test request is processing correctly. And errors are displayed to the user.

6. Performance Testing:

- Performance testing ensures that the web application functions properly under high demand. Web performance testing is divided into two categories: Web Stress Testing and Web Load Testing.
- This includes to:
  - Test the response time of the Website application at various connection speeds.
  - Determine if the site can manage a large number of concurrent user queries.
  - Examine how your web application performs during high traffic.
  - Test using a huge user input data.
  - Examine the behavior of the web application if several connections to the database are made.
  - Examine how the website performs if a crash happens due to peak traffic.
  - Examine optimization strategies such as reducing load times by activating caching on both the client and server sides of the browser, compression, and so on.
  - Check for any hardware memory leakage problems.

7. Security Testing:

- Security testing is carried out to ensure that no information is leaked while encrypting the data.
- Security testing is essential for e-commerce websites. Check on the security-sensitive information such as credit cards, etc. can be carried out in the security testing.
- Security testing activities will include:
  - Check for unauthorized access to secured sites; if the user switches from "https" to "http" (secure to non-secure) for secured pages, the appropriate notice should be displayed, and vice versa.
  - Determine if the user is visiting internal sites directly by typing URLs into the browser. If authentication is necessary, the user should be forwarded to the login page or an appropriate notification should be provided.

- The log file should contain the majority of the information relating to transactions, error messages, and login attempts.
- Determine if internal Web directories or files are inaccessible unless and until they are set for download.
- Check if CAPTCHA is added and is working properly for login to prevent automatic login attempts.
- Test for accessing other people's information by modifying a query string parameter. For example, if you're modifying the information and you see UserID = 123 in the URL, try changing this parameter value and seeing if the application still doesn't provide the other useful information. It should show that this user does not have permission to read other users' information.
- Determine if a session has expired after a pre-defined length of time if it has been idle for that period of time.
- Determine if the user is unable to proceed to the login screen due to an incorrect username/password combination.

Table 5.1: Client/Server and Web based Testing [S-22, 23; W-22]

Sr. No.	Client/Server	Web-based Testing
1.	Tester tests two components: client-side and server-side.	In this testing, the tester does not have control over the application.
2.	Application is loaded on the server and its executable file is tested on every client machine.	Application is tested on different browser and OS platforms.
3.	Tester is aware of some clients and server and their location and should be included in the test scenario.	Tester does not have control over the application. Tester has to test it on different web browsers.
4.	Testing elements are GUI on the client and server-side, load testing, functionalities, and client-server interaction.	Here testing elements are browser compatibility, OS compatibility, error handling, static pages, backend testing, and load testing.

5.4 TESTING DOCUMENTATION AND HELP FACILITIES

Documentation testing is a type of non-functional software testing. Any written or graphic information that describes, defines, specifies, or certifies actions, is tested. Documentation is just as crucial as the product itself in terms of success. If the documentation is weak, non-existent, or incorrect, it reflects poorly on the product and the manufacturer.

According to IEEE, documentation types include test case specification, test incident report, test log, test plan, test method, and test report, which describe plans for or

outcomes of testing of a system or component. As a result, testing of all of the above-mentioned papers is referred to as documentation testing.

- This is one of the most cost-effective methods of testing. If the documentation is incorrect, there can be severe consequences. The documentation may be evaluated in a variety of ways with varying degrees of sophistication. It might range from using a spelling and grammar checker to personally examining the documentation to remove any ambiguity or inconsistency.
- Documentation testing may begin at the very beginning of the software process, saving significant money because the earlier a flaw is discovered, the less it will cost.
- Defects in documentation can be just as harmful to software acceptance as errors in data or source code. Nothing is more annoying than precisely following a user guide or an online help facility and receiving outcomes or behaviors that do not correspond to those indicated by the documentation. As a result, documentation testing should be an important element of any software test plan.
- Documentation testing can be performed in two phases. The first phase includes review and inspection, which examines the document for editorial clarity. The second phase includes a live test, that uses the documentation in conjunction with the use of the actual program.
- A live test for documentation may be addressed in the same way that many black-box testing approaches are. Graph-based testing may be used to explain how the program is used, while equivalence partitioning and boundary value analysis can be used to identify different types of input and related interactions. The documentation is then used to track program usage.
- The following questions should be answered during both phases:
  - Does the documentation correctly describe how to use each mode of operation?
  - Is the description of each interaction sequence accurate?
  - Are the examples correct?
  - Do the vocabulary, menu descriptions, and system replies match the actual program?
  - Is it simple to find assistance within the documentation?
  - Is it possible to debug using the documentation?
  - Is the document's table of contents and index correct and up to date?
  - Is the document's design (layout, fonts, indentation, and visuals) favorable to comprehension and speedy information acceptance?
  - Are all software error messages provided to the user in the document mentioned in further detail? Are the actions to be done as a result of an error notice clearly defined?

- Are hypertext links used, and if so, are they correct and complete?
- Is the navigation architecture adequate for the information necessary if hypertext is used?

5.4.1 User Documentation Testing

[S-22]

- User documentation includes: user manuals, user guides, installation and set-up instructions, online help, read me file(s), software release notes, and many more documents that are offered to end-users with the program to assist them to understand the software system.
- Inadequate documentation might lead to incorrect system changes or wrong use of system output. Both of these issues might result in inaccurate system output. "Documentation testing" is the recommended test technique for documentation.
- User documentation testing has two objectives:
  - To determine whether the information in the document is available in the product.
  - To ensure that what is in the product is properly explained in the documentation.
- When a product is upgraded, the corresponding product documentation should also get updated simultaneously to avoid defective documentation.
- As previously mentioned, user documentation focuses on ensuring that what is in the document perfectly matches the product behavior, which is done by validating screen by screen, transaction by transaction, and report by report. Documentation testing also examines the document's linguistic features, such as spell check and grammar.
- Defects found in user documentation need to be tracked properly along with the other defect information such as Defect/comment description, Paragraph/page number, Document version number reference, name of the Reviewer, Name of author, Priority, and Severity of the defect.
- This defect information is passed to the corresponding author for defect fixing and closure of the defect. Because these documents are the first interactions the users have with the product. Good User Documentation aids in reducing customer support calls. The time and money invested in this effort would be a good investment for the company in the long term.
- This testing plays a very important role as the user will refer to this document when they start using the software at their location. A poor document can put off a user and bias them against the product even the product offers rich functionality.

5.4.2 Benefits of User Documentation Testing

1. User documentation testing helps in identifying the problems unnoticed during reviews.

2. Good documentation guarantees the reliability and integrity of documentation and product, reducing the number of potential problems reported by consumers.
3. It minimizes support expenses by shortening the time required for each support contact by pointing consumers to the relevant portion of the handbook.
4. New programmers and testers can utilize the documentation to learn about the product's external capability.
5. Customers require less training and may begin using the product sooner. As a result, high-quality documentation can help firms cut total training expenses.

5.5 TESTING FOR REAL-TIME SYSTEMS [S-22, 23; W-22]

- The time-dependent, asynchronous nature of many real-time applications adds a new and potentially difficult element to the testing mix-time. The test case designer must consider not only white-box and black-box test cases, but also event management (i.e., interrupt processing), data timeliness, and the parallelism of the tasks (processes) that handle the data. In many cases, providing test data when a real-time system is in one state will result properly, whereas providing the same data when the system is in another state may result in an error.
- Air Traffic Control Systems, Networked Multimedia Systems, Command Control Systems, real-time photocopying, etc are typical examples for real-time system testing.
- Different conditions cause real-time systems to respond differently.
- In addition, the intimate relationship that exists between real-time software and its hardware environment can also cause testing problems. Software tests must consider the impact of hardware faults on software processing.
- Such faults can be extremely difficult to simulate realistically.

Process of Real-time Software:

- Comprehensive test case design methodologies for real-time systems are still in the early stages of development. However, there is a four-step technique that is suggested:

1. **Task Testing**: The first step is to test each task separately. For each assignment, white-box and black-box tests are designed and executed. During these tests, each task is completed individually. Task testing reveals only logical and functional flaws; timing and behavior aspects are not covered.
2. **Behavioral Testing**: System models built using CASE tools are referred to as behavioral testing as they are used to simulate the behavior of a real-time system and are assessed as a result of external events. These activities may be used to create test cases that will be run once the real-time software has been completed.
   - Events (e.g., interruptions, control signals) are grouped for testing using an approach similar to equivalence partitioning. Example events for the photocopier examples might be - user interrupts (e.g., reset counter), mechanical interrupts

(e.g., paper jammed), system interrupts (e.g., toner low), and failure modes (e.g., roller overheated).

- Each of these events is evaluated separately, and the executable system's behavior is checked for faults that occur as a result of the processing connected with these events. Compliance may be determined by comparing the behavior of the system model (produced during the analysis activity) and the executable program.
- Events are delivered to the system in random order and with random frequency once each kind of event has been examined. The software's behavior is evaluated to discover behavior problems

3. **Intertask Testing:** Once errors in individual tasks and system behavior are done, the testing process is shifted to time-related errors. Intertask synchronization faults are assessed using various data rates and processing loads on asynchronous jobs. Tasks that interact via a message queue or data store are also examined to identify size issues in these data storage regions.
4. **System Testing:** To find faults at the software/hardware interface, software and hardware are integrated and a broad variety of system tests are performed. Interrupts are handled by the majority of real-time systems.

The tester creates a list of all possible disturbances and the processing that occurs as a result of the disturbances using the state transition diagram and the control specification.

Then tests are created to evaluate the following system characteristics:

- Is interrupt priority allocated and handled correctly?
- Is each interrupt's processing handled correctly?
- Does each interrupt-handling procedure's performance (e.g., processing time) meet the requirements?
- Is a high amount of interrupts occurring at key moments causing performance or function issues?

Furthermore, global data areas that are used to transfer the information as part of interrupt processing should be evaluated to see if they get the potential to cause side effects.

Summary

- Graphical User Interfaces (GUIs) present interesting challenges for software engineers.
- Standards/guidelines, intuitiveness, consistency, adaptability, comfortability, accuracy, and usability are seven fundamental elements of a successful user interface.
- Windows compliance and web-app interface testing are two types of GUI testing.
- Interface testing techniques such as links, forms, client-side scripting, pop-up windows, cookies, and a GUI checklist for validation, aesthetic, and navigation requirements are included in web app interface testing.

- Functionality, integration, and performance testing, as well as component testing, integration testing, performance testing, concurrency testing, disaster recovery, and compatibility testing, are all part of a Client/Server test strategy.
- User documentation testing, which includes user manuals, user guides, installation, and set-up instructions, helps in detecting errors unnoticed during reviews.

Check Your Understanding

1. A complete review of the interface design model is done using _____.
   (a) Cookies (b) Standards
   (c) GUI checklist (d) None of these
2. _____ is necessary to understand how a system reacts when multiple users are accessing the same data at the same time.
   (a) Recovery testing (b) Concurrency testing
   (c) Compatibility testing (d) GUI testing
3. _____ are active as long as the browser associated with that cookie is open.
   (a) Session cookies (b) Web forms
   (c) Databases (d) Persistent cookies
4. Which of the following is applicable to compatibility testing?
   (a) Functional aspect (b) Non-functional aspect
   (c) Links (d) Data reliability
5. _____ testing is carried out to ensure that no information is leaked while encrypting the data.
   (a) Navigation (b) Operating system
   (c) Security (d) None of these.
6. Functionality testing determines whether each _____ works as planned.
   (a) navigation (b) hardware
   (c) component (d) API
7. The first phase of documentation testing which examines the document for editorial clarity includes _____.
   (a) review (b) web-based testing
   (c) inspection (d) Both (a) and (c)
8. Which of the following tests the application loaded on the server and its executable file on every client machine?
   (a) Web application testing (b) Performance testing
   (c) Browser testing (d) Client/Server testing

9. Which of the following is not the process of real-time software?
   (a) Intertask testing (b) Task testing
   (c) Behavioral testing (d) None of these
10. Which of the following is a part of GUI checklist?
   (a) Disaster Recovery (b) Cookies
   (c) Navigation testing (d) Load testing

Answers

1. (c)	2. (b)	3. (a)	4. (b)	5. (c)
6. (c)	7. (d)	8. (d)	9. (d)	10. (c)

Practice Questions

Q.I Answer the following questions in short.

1. Which are the GUI testing types?
2. What is web-based testing?
3. What is user documentation testing?
4. Which testing technique is required for the Client/Server testing process?
5. Which is the first step in real-time software testing?

Q.II Answer the following questions.

1. Explain GUI testing in detail.
2. With the help of a diagram describe Client/Server testing.
3. Which are tests conducted For Client/Server architecture?
4. Which are testing methods are used for a Real-time system?
5. Explain in detail the testing for User Documentation.
6. Which are the benefits of User Documentation testing?
7. What is the difference between Client/Server and Web-Based testing?
8. Explain in detail the Web-Based testing?

Q.III Define the terms.

1. System testing
2. Behavioral testing
3. Performance testing
4. Security testing
5. GUI checklist

★★★

6...

Testing Tools and Software Quality Assurance (Introduction)

Learning Objectives ...

- To study software testing tools such as Junit, Apache JMeter, Winrunner, Loadrunner, Rational Robot
- To understand the concept of Quality and Quality movements and Background issues.
- To learn about Software Quality Assurance Activities.
- To get information about Software Reliability.
- To know about ISO 9000 Quality Standards.
- To study SQA plan, Six Sigma Methodologies, and Informal Reviews.

6.1 INTRODUCTION [S-22]

- Software testing tools are applications that developers and testers can use to help them perform manual and automated tests. Unit testing, integration testing, regression testing, end-to-end testing, performance testing, compliance testing, and security testing are all performed by different tools.
- Software Quality Assurance (SQA) is a continuous process that assures that the software product meets and exceeds the organization's standardized quality criteria. SQA is a set of actions that ensures that everyone working on the project has followed all procedures and protocols.

6.2 JUNIT, APACHE JMETER, WINRUNNER

6.2.1 JUnit

- JUnit is an open-source testing framework for the Java programming language. It is important for test-driven development and is used for testing a small chunk of code or unit.

- It is used to verify a small chunk of code by creating a path, function, or method.
- JUnit encourages the concept of "first testing, then coding," highlighting the importance of setting up test data for a piece of code that may be tested first and then implemented.
- This method improves program code productivity and stability, which minimizes programmer stress and troubleshooting time.
- Tester/developer can use JUnit to ensure that methods in the Java code perform as expected without having to build up the entire program. The minimum supported Java version from JUnit 4.0 onwards is 1.5.0. There are special extensions for implementing the JUnit testing in specific domains like XML or databases.
- Sample JUnit Testing Tool's Screen is stated in figure 6.1.

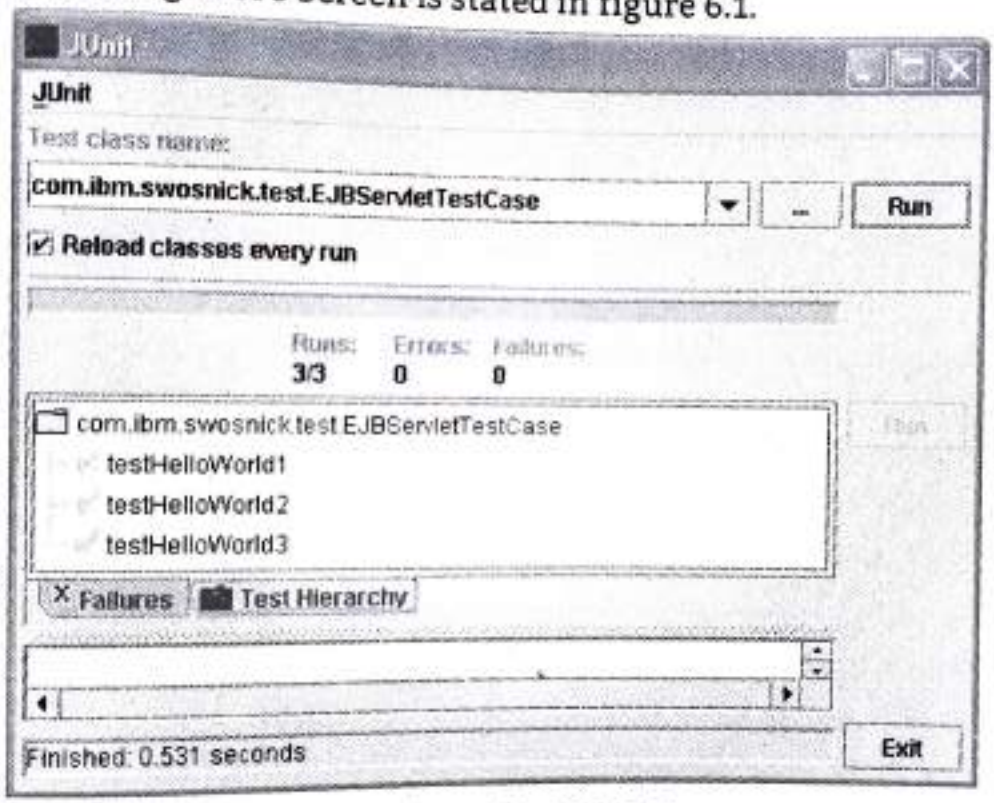


Fig 6.1: JUnit testing Tool's Screen

6.2.1.1 Features of JUnit

- JUnit is a free and open-source framework for creating and running tests.
- It has annotations to help you identify the test methods.
- It includes Assertions that can be used to verify expected outcomes.
- It has Test Runners that may be used to conduct tests.
- JUnit tests enable you to write code more quickly, which improves quality.
- JUnit is elegantly simple, requires less time, and is less sophisticated.
- JUnit tests can be run automatically, and they may examine and provide quick feedback on their results. There's no need to go through a report of test findings by hand.
- JUnit tests can be combined into test suites that include test cases and possibly other test suites.

- Junit displays test progress (as a green bar if everything is going well and a red bar if the test fails).

6.2.1.2 Reasons for Using JUnit

1. **JUnit tests make it possible to write quality code more quickly:** JUnit tests enable you to write code rapidly and identify errors quickly.
2. **JUnit is a simple and free unit testing framework**:You can rapidly develop tests that examine your components' interfaces with JUnit. Running tests with JUnit is as simple and quick as compiling your code.
3. **JUnit tests check and provide quick feedback on their results**: JUnit tests can run on their own and check their results. When you run tests, you receive immediate and straightforward feedback on whether they passed or failed. There's no need to go revisit a pile of test results to find what you're looking for.
4. **JUnit tests can be arranged logically into test suites that can be executed automatically**:This enables automated regression testing of a set of tests to arbitrary depths.
5. **Writing JUnit tests is a simple task :** It is simple to write JUnit tests. You can write tests with the JUnit testing framework while also taking advantage of the framework's testing environment. Writing a test is as simple as specifying the desired outcome and writing a function that exercises the code to be tested.
6. **JUnit tests increase the stability of the software**: The fewer tests you write the less stable your code becomes. Tests validate the stability of the software and instill confidence that changes haven't caused a ripple effect through the software. The tests form the glue of the structural integrity of the software.
7. **JUnit tests are developer tests:** JUnit tests are highly localized tests that are created to help developers increase their productivity and code quality. Unit tests are created to test the core building elements of the system from the inside out, as opposed to functional tests, which treat the system as a black box and guarantee that the software works as a whole. When development iteration is complete, the tests are promoted as a part of the delivered component as a way of communicating, "Here's my deliverable and the tests which verify it."
8. **JUnit tests are written in Java:** Using Java tests to test Java software creates a smooth connection between the test and the code under test. The Java compiler aids the testing process by evaluating the unit tests for static syntax and ensuring that the software interface contracts are followed.
9. **JUnit provides different test runners:** JUnit provides three test runners: a text-based, an AWT-based, and a Swing-based.

6.2.1.3 Common Terms used in JUnit

- Some common terms used in JUnit are:
  - **Test method:** A method in a Java class that contains a single unit test.
  - **Test class:** A Java class containing one or more test methods.
  - **Assertion:** A statement that you include in a test method to check that the results of a test are as expected.
  - **Test fixture:** A class that is used to set up a state for multiple tests. Typically it is used when the setup routines are "expensive" or take a long time to execute.
  - **Test suite:** A grouping of test classes that are run together.

6.2.1.4 Advantages and Disadvantages of JUnit

Advantages:

- It is a small framework with a lot of documentation.
- It improves the testing process and make it easier to execute the tests.
- Most Java development tools, including BlueJ, are supported.
- Testing becomes more systematic and, in certain cases, automated.
- Unit testing Java programs has become the de facto standard.

Disadvantages:

- Java-specific but being ported to another language.
- It's not designed to handle huge test suites.

6.2.2 Apache JMeter

- Apache JMeter is a free and open-source testing tool. It is a load and performance testing program written entirely in Java.
- It was developed by Stefano Mazzocchi of the Apache Software Foundation. He wrote it solely to evaluate Apache Tomcat's performance. JMeter was later modified by Apache to improve the user interface and include functional testing capabilities.
- JMeter is a graphical Java desktop application that makes use of the Swing graphical API. As a result, it can run on any environment/workstation that supports a Java virtual machine, such as Windows, Linux, Mac, etc.

6.2.2.1 Applications

1. JMeter is designed to cover various categories of tests such as load testing, functional testing, performance testing, regression testing, etc. It requires JDK 5 or higher version.
2. Apache JMeter may be used to test performance both on static and dynamic resources (files, Servlets, Perl scripts, Java Objects, Data Bases and Queries, FTP Servers, and more).

3. It can be used to test a server's, network's, or object's strength by simulating a significant demand on it.
4. It can be used to create a performance graph.

6.2.2.2 Protocols Supported by JMeter

- The protocols supported by JMeter are:
  - Web – HTTP, HTTPS sites 'web 1.0' web 2.0 (Ajax, flex and flex-ws-amf)
  - Web Services – SOAP / XML-RPC
  - Database via JDBC drivers
  - Directory – LDAP
  - Messaging oriented service via JMS
  - Service – POP3, IMAP, SMTP
  - FTP Service

6.2.2.3 Features of JMeter

- Following are some of the features of JMeter:
  - Being open-source software, it is freely available.
  - It has a simple and intuitive GUI.
  - JMeter can conduct load and performance tests for many different server types - Web - HTTP, HTTPS, SOAP, Database via JDBC, LDAP, JMS, Mail - POP3, etc.
  - It is a platform-independent tool. On Linux/Unix, JMeter can be invoked by clicking on the JMeter shell script. On Windows, it can be invoked by starting the jmeter.bat file.
  - It has full swing and lightweight component support (precompiled JAR uses packages javax.swing.*).
  - JMeter stores its test plans in XML format. This means you can generate a test plan using a text editor.
  - Its comprehensive multi-threading framework allows several threads to test at the same time and various thread groups to test different functions simultaneously.
  - It is highly extensible and can also be used to perform automated and functional testing of the applications.

6.2.2.4 How does JMeter Works?

- JMeter simulates several users sending requests to a target server and returns statistics in the form of tables, graphs, and other visual representations of the target server's/performance/functionality. application's

Take a look at Figure 6.2, which demonstrates how JMeter works:

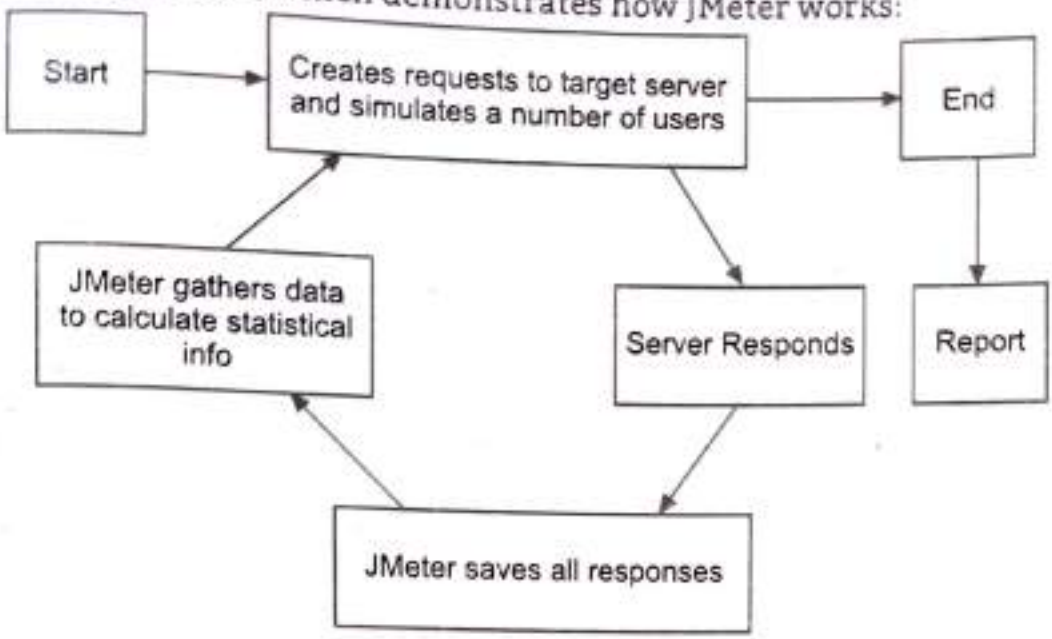


Fig. 6.2: Working of JMeter

- JMeter is not a browser. As far as web services and remote services are concerned, JMeter looks like a browser (or rather, multiple browsers); however, JMeter does not perform all the actions supported by browsers.
- In particular, JMeter does not execute Javascript found in HTML pages, nor does it render HTML pages in the same way that a browser does

6.2.3 WinRunner

- WinRunner is the most used Automated Software Testing Tool. HP/Mercury Interactive WinRunner is an automated functional GUI testing tool that allows a user to record and playback UI interactions as test scripts.

6.2.3.1 Features of WinRunner

- It was developed by Mercury Interactive.
- It is a tool for testing functionality.
- It supports Client/Server and web technologies such as VB, VC++, D2K, Java, HTML, Power Builder, Delphi.
- Quick test professionals can be used to support.net, XML, SAP, Peoplesoft, Oracle applications, and multimedia.
- WinRunner run on Windows OS only.
- X-Runner runs only on UNIX and Linux.
- Tool developed in C on VC++ environment.
- WinRunner used TSL to automate the test (Test Script language like C).

6.2.3.2 Testing Process in WinRunner

- The main testing process in WinRunner is:
  1. **Learning:** Recognition of objects and windows in our application by WinRunner is called learning. WinRunner 7.0 follows Auto learning.

2. **Recording:** WinRunner records the manual business operation in TSL.
3. **Edit Script:** It depends on the corresponding manual test. Test engineer inserts checkpoints into that record script.
4. **Run Script:** During the execution of the test script, WinRunner compares the tester's expected values to the application's actual values and displays the results.
5. **Analyze Result:** Tester analyzes the tool given results to concentrate on defect tracking if required.

6.2.3.3 Recording Modes of WinRunner

- It supports the following two types of recording modes:
  1. **Context-Sensitive Recording:** Context-Sensitive mode records user actions on the application under test in terms of selected GUI objects like windows, lists, buttons, etc. while ignoring the actual location of the object on the screen. Every time any action is done on the application under test, a TSL statement gets generated in the test script. This TSL statement describes the details of the object selected and the action performed.
  2. **Analog Recording:** Analog mode records mouse clicks, keyboard inputs, and the precise two-dimensional (X, Y) coordinates traversed by the mouse. When the test is run, WinRunner retraces all the mouse tracks. When exact mouse coordinates are important to the test, an analog mode is used, for example, testing a drawing application.

6.2.3.4 How to execuit the Tests?

- The WinRunner performs a line-by-line interpretation. During the test, WinRunner runs the application as if someone is physically present at the controls.
- WinRunner has three different run modes.
  1. **Verify mode:** This is used to test the application.
  2. **Debug mode:** This is used to debug the test.
  3. **Update mode:** This is used to refresh the desired results.

- **Debugging Tools:**
  - When a test stops in between an execution due to some syntax error or some or logic error, we can use several methods and tools to identify and isolate the problem.
    1. **Use of Step commands:** This is used to run a single line or a specific section of a test.
    2. **Use of Breakpoints:** This is done to pause the test run at pre-determined points, to enable us to identify the flaws in the test.
    3. **Use of the Watch List:** This is done to keep track of the variables, expressions, and array elements in the test. During a test's execution, we can see the values at each breakpoint, such as after the step command, some breakpoints, or the end of the test.

6.3 LOADRUNNER, RATIONAL ROBOT

6.3.1 LoadRunner

[S-22, 23; W-22]

- LoadRunner enables system testing under controlled and peak load conditions. LoadRunner generates load by running thousands of Virtual Users distributed across a network. These Virtual Users provide consistent, repeatable, and measurable load to exercise your application in the same way that real users would.
- LoadRunner's detailed reports and graphs provide the data you need to assess the performance of your application. LoadRunner simulates a scenario in which thousands of users interact with a Client/Server system at the same time. LoadRunner accomplishes this by substituting a virtual user for the human user (Vuser). A Vuser Script describes the actions that a Vuser takes.

6.3.1.1 Advantages of LoadRunner over Manual Performance Testing

- LoadRunner reduces personnel requirements by substituting virtual users(Vusers), for human users. These Vusers mimic real-world user behavior by running real-world applications. LoadRunner reduces hardware requirements by allowing multiple Vusers to run on a single computer.
- The LoadRunner Controller enables you to control all of the Vusers from a single point of control.
- During a test, LoadRunner automatically records the application's performance. To view the performance data, you can select from a variety of graphs and reports. It investigates the causes of performance delays, such as network or client delays, CPU performance, I/O delays, database locking, or other issues at the database server.
- LoadRunner monitors network and server resources to assist you in optimizing performance. LoadRunner tests are fully automated, so you can easily repeat them as many times as you need.

6.3.1.2 Most Commonly used terms in the LoadRunner

- **Scenarios:** Using LoadRunner, you can divide the application performance testing requirements into scenarios. A scenario defines the events that occur during each testing session. Thus, for example, a scenario defines and controls the number of users to emulate the actions that they perform, and the machines on which they run their emulations.
- **Vusers:** In the scenario, LoadRunner replaces human users with virtual users or Vusers. When you run a scenario, Vusers emulate the actions of human users working with the application. While a workstation accommodates only a single human user, many Vusers can run concurrently on a single workstation. A scenario can contain tens, hundreds, or even thousands of Vusers.
- **Vuser Scripts:** The actions that a Vuser performs during the scenario are described in a Vuser script. When you run a scenario, each Vuser executes a Vuser script. The

Vuser scripts include functions that measure and record the performance of your application's components.

- **Transactions:** Transactions are used to measure the server's performance. A transaction represents an action or set of actions that you want to detect. Within the Vuser script, the testers define transactions by enclosing the appropriate sections of the script with start and end transaction statements. For example, a transaction can be defined to measure the server's response time to view a user's account balance and the information to be displayed at the ATM.
- **Rendezvous points:** You insert Rendezvous points into Vuser scripts to emulate heavy user load on the server. Rendezvous points instruct Vusers to wait during test execution for multiple Vusers to arrive at a certain point, so that they may simultaneously perform a task. For example, to emulate peak load on the bank server, you can insert a rendezvous point instructing 100 Vusers to deposit cash into their accounts at the same time.
- **Controller:** The LoadRunner Controller is used to manage and maintain scenarios. Using the Controller, you can control all the Vusers in a scenario from a single workstation.
- **Load generator:** When you execute a scenario, the LoadRunner Controller distributes each Vuser in the scenario to a load generator. The load generator is the machine that executes the Vuser script, enabling the Vuser to emulate the actions of a human user.
- **Performance analysis:** Vuser scripts include functions that measure and record system performance during load-testing sessions. During a scenario run, you can monitor the network and server resources. Following a scenario run, you can view performance analysis data in reports and graphs.

6.3.1.3 LoadRunner Vuser Types

- LoadRunner has various types of Vusers. Each type is designed to handle different aspects of today's system architectures. You can use the Vuser types in any combination in a scenario to create a comprehensive application test. The following Vuser types are available:
- **Client/Server:** For MS SQL Server, ODBC, Oracle (2-tier), DB2 CLI, Sybase Ctlib, Sybase Dblib, Windows Sockets, and DNS protocols.
- **Custom:** For C templates, Visual Basic templates, Java templates, JavaScript, and VBScript typescripts.
- **Distributed Components:** For COM/DCOM, Corba-Java, and RMI-Java protocols.
- **E-business:** For FTP, LDAP, Media Player, Multi-Protocol Web/WS, Web (HTTP, HTML), Palm, and RealPlayer protocols. ERP: For Oracle NCA, PeopleSoft (Tuxedo), SAP, and Siebel protocols.
    - **Legacy:** For Terminal Emulation (RTE). Mailing Services: Internet Messaging (IMAP), MS Exchange (MAPI), POP3, and SMTP.
    - **Middleware:** For the Tuxedo (6, 7) protocol.
    - **Wireless:** For i-Mode, VoiceXML, and WAP protocols.

6.3.1.4 The LoadRunner Testing Process

- By following the LoadRunner testing procedure listed below and outlined in figure 6.3, testers can easily create and run load-test scenarios.

Step I: Planning the Test:

- A clearly defined test plan will ensure that the LoadRunner scenarios will accomplish load-testing objectives.

Step II: Creating the VuserScripts:

- Vusers emulate the actions of human users by performing typical business processes in your Web-based application. A Vuser script contains the actions that each virtual user performs during scenario execution. In each Vuser script, you determine the tasks that will be:
  - Performed by each Vuser.
  - Performed simultaneously by multiple Vusers.
  - Measured as transactions LoadRunner.

Step III: Creating the Scenario:

- A scenario describes the events that occur during a testing session. A scenario includes a list of machines on which Vusers run, a list of scripts that the Vusers run, and a specified number of Vusers or Vuser groups that run during the scenario.
- You create scenarios using the LoadRunner Controller. Creating a manual scenario you create a scenario by defining Vuser groups to which you assign several individual Vusers, Vuser scripts, and load generators to run the scripts.
- You can also create a scenario using the Percentage Mode, in which you define the total number of Vusers to be used in the scenario, and the load generator machines and percentage of the total number of Vusers to be assigned to each Vuser script.
- For Web tests, you can create a goal-oriented scenario in which you define the objectives you want your test to achieve. LoadRunner will generate a scenario for you based on these objectives.

Step IV: Running the Scenario:

- By ordering numerous Vusers to complete tasks at the same time, you can simulate user load on the server. Increase or decrease the number of Vusers performing tasks at the same time to adjust the load level. Before running a scenario, you must first configure it and schedule it. When you execute the scenario, this controls how all of the load generators and Vusers act.
- You can run the entire scenario, groups of Vusers (Vuser groups), or individual Vusers. While a scenario runs, LoadRunner measures and records the transactions that you defined in each Vuser script. You can also monitor the system's performance online.

Step V: Monitoring a Scenario:

- Scenario execution can be monitored using the LoadRunner online run-time, transaction, system resource, Web server resource, Web application server resource, database server resource, network delay, streaming media resource, firewall server resource, ERP server resource, and Java performance monitors.

Step VI: Analyzing Test Results:

- During scenario execution, LoadRunner logs the application's performance under various loads. The application's performance can be analyzed using LoadRunner's graphs and reports.

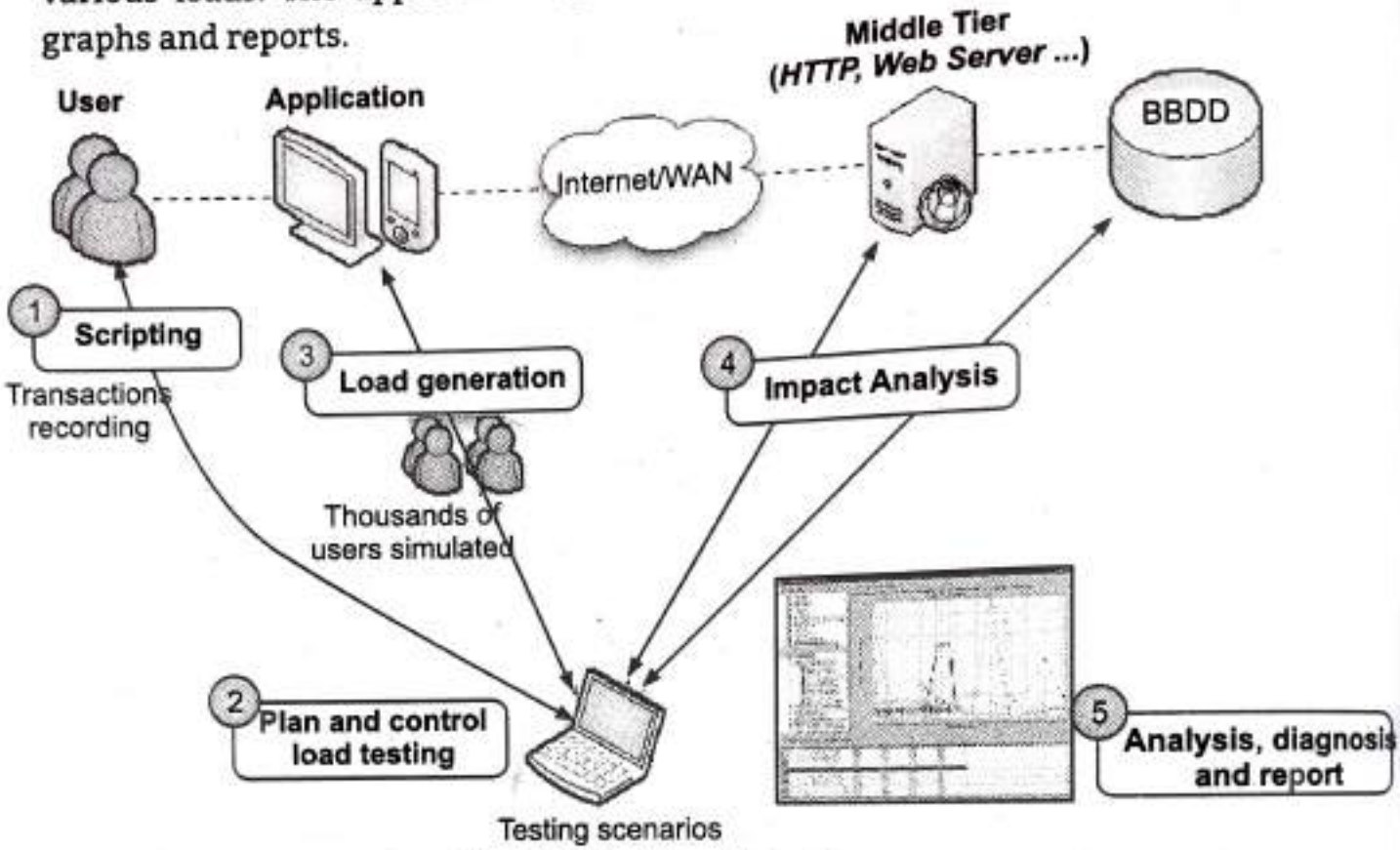


Fig. 6.3: Working of LoadRunner

6.3.2 Rational Robot

- A Rational Robot is a software that automates the testing process. It contains test cases for items like lists, menus, and bitmaps. For eCommerce, Client/Server, and ERP systems, IBM Rational Robot automates regression, functional, and configuration testing.
- It is used to test applications based upon a wide variety of user interface technologies. It is integrated with the IBM Rational TestManager solution to provide desktop management support for all testing activities.
- It identifies defects, offers test cases and test management, and works with a variety of user interface technologies.
- It gives Quality Assurance teams a general-purpose test automation tool for functional testing of Client/Server applications.
- It reduces the time it takes for testers to see the value of test automation methods.
- It enables test-automation engineers to detect defects by extending test scripts and defining test cases.
- It provides test cases for common objects and a specialized test case to development environment objects includes built-in test management, integrates with IBM Rational Unified Process tools.

- It helps in defect tracking, change management, and requirements traceability.
- It supports multiple UI technologies.
- Operating systems supported: Windows family.

6.3.2.1 Components of Rational Robot

- **Rational Administrator** : used to create and manage rational projects to store the testing information.
- **Rational TestManager** : used to review and analyze test results.
- **Object Properties, Text, Grid, and Image Comparators** : used to view and analyze the results of verification point playback.
- **Rational Site Check** : used to manage Internet and Intranet websites.

6.3.2.2 Working of Rational Robot

- Rational Robot allows all members of the development and testing teams to implement a complete and effective testing methodology.
- Robot automates your testing effort by replacing the laborious, frequently error-prone practice of manual testing with software solutions.
- You save time and guarantee that your testing process generates predictable and accurate results using Robot's automated functional testing.
- With a few mouse clicks, you can begin recording the tests. Robot replays the tests in a fraction of the time it would take to repeat the tasks manually after recording them.
- Robot's Object-Oriented Recording technology lets you generate scripts quickly by running the application-under-test. Object-Oriented Recording identifies objects by internal object names, not by screen coordinates. And, if objects in successive builds change locations or their text changes, Robot still finds and tests them during playback.
- With Object Testing, testers can test hundreds, even thousands of properties for an object as well as an object's data.

6.3.2.3 Features of Rational Robot

- **Simplifies configuration testing** : Rational Robot can be used to distribute functional testing among many machines, each one configured differently. The same functional tests can run simultaneously, and so reduces the time to identify problems with specific configurations.
- **Tests many types of applications** : Rational Robot supports a wide range of environments and languages, including HTML and DHTML, Java™, .NET, Microsoft Visual Basic and Visual C++, Oracle Developer/2000, Sybase PowerBuilder, and Borland Delhi.
- **Ensures testing depth** : With a single mouse click, you may test hundreds of characteristics of an application's component objects, such as ActiveX Controls, OCXs, Java applets, and many others.

- **Tests custom controls and objects :** Rational Robot allows you to test each application component under varying conditions and provides test cases for menus, lists, alphanumeric characters, bitmaps, and many more objects.
- **Provides an integrated programming environment :** Rational Robot generates test scripts in SQA Basic, an integrated MDI scripting environment that allows you to view and edit your test script while you are recording.
- **Helps to analyze problems quickly :** Rational Robot stores test findings in the Rational Repository and color tag them for easy visual examination. By double-clicking on an entry, you are brought directly to the corresponding line in the test script, in that way ensuring fast analysis and correction of test script errors.
- **Enables reuse :** Rational Robot ensures that the same test script, without any modification. It can be reused to test an application running on Microsoft Windows.

6.3.2.4 Applications

- Rational Robot is used to:
  - Perform full functional testing. Record and playback scripts that navigate through the application and test the state of objects through verification points.
  - Perform full performance testing. Use Robot and TestManager together to record and playback scripts that help you determine whether a multi-client system is performing within user-defined standards under varying loads.
  - Create and edit scripts using the SQA Basic, VB, and Virtual User scripting environments: The Robot editor provides color-coded commands with keyword Help for powerful integrated programming during script development.
  - Test applications developed with IDEs such as Visual Studio.NET, Visual Basic, Oracle Forms, PowerBuilder, HTML, and Java. Test objects even if they are not visible in the application's interface.
  - Collect diagnostic information about an application during script playback. The robot is integrated with Rational Purify, Quantify, and PureCoverage. You can playback scripts under a diagnostic tool and see the results in the log.

6.3.2.5 Types of Scripts

- The robot records several types of scripts such as,
  - **OSQA Basic scripts (using MS-Basic language syntax):** These scripts capture the commands equivalent to each user action.
  - **RobotJ scripts (using Java language syntax):** These scripts are compiled into .class files containing Java bytecode.
  - **Virtual User (VU) scripts (using C language syntax):** These scripts capture entire streams of conversations HTML, SQL, Tuxedo, CORBA Inter-ORB IIOP, and raw Sockets Jolt protocols sent over the network. These are compiled into dynamic-link library (.dll) files and linked into an a.obj compiled from a.c source file which calls the .dll file.

6.4 QUALITY CONCEPTS, QUALITY MOVEMENT, BACKGROUND ISSUES, SQA ACTIVITIES

6.4.1 Quality Concepts

- Quality means consistently meeting client expectations in terms of service, delivery time, and cost.
- Quality software is bug-free, on time, within budget, satisfies client needs, and is easy to maintain.
- Software quality is also measured by three criteria:
  1. Failures.
  2. Reliability.
  3. Customer satisfaction.
- A software product is said to have good quality if:
  - It has a few post-release failures.
  - It is reliable, meaning that it hardly crashes or behaves unexpectedly when used by the customer.
  - It keeps a majority of customers satisfied and happy.

Types of quality:

- There are two kinds of quality:
  1. **Quality of design:** It refers to the qualities that designers specify for the finished product to be built.
     It includes requirements, specifications, and the design of the system.
  2. **Quality of conformance:** The degree to which the product's design specifications are fulfilled during production. It focuses mostly on implementation.

Definitions of Quality:

- "Quality consists of all those product features which meet the needs of customers and thereby provide product satisfaction. Quality consists of freedom from deficiencies".
  - Juran
- "A predictable degree of uniformity and dependability at low cost and suited to the market".
  - Dr. Edward Deming

6.4.2 Quality Assurance (QA)

- QA focuses on the product's manufacturing process to prevent problems.
- The idea is to give customers trust that a product will meet their expectations.
- QA procedures include auditing and reporting, which are designed to give management the information they need to make proactive decisions.
- If there is a problem in the data provided through quality assurance, it is management's responsibility to address the problems and apply the necessary resources to resolve them.

6.4.3 Quality Control (QC)

- QC aims to identify defects in the final product.
- It focuses on operational techniques and the activities used to fulfill and verify quality requirements.
- Inspections, reviews, and tests are done throughout the software process to ensure that each product meets its specifications/requirements.
- The key concept of quality control is "all work products have defined and measurable specifications", which are compared with the output of each process.
- Table 6.1 shows the difference between QC and QA.

Table 6.1: QC Vs QA

Quality Control – QC	Quality Assurance - QA
It is product oriented.	It is a process oriented.
QC aims to identify defects in the final product. (It is a reactive process)	QA aims to prevent defects with a focus on the process used to make the product. (It is a proactive quality process)
The goal of QC is to identify defects after a product is developed and before it's released. (Find the defects)	The goal of QA is to improve development and test processes so that defects do not arise when the product is being developed. (Prevent the defects)
Focuses on the activities or techniques used to achieve and maintain the product quality, process, and service.	Focuses Prevention of quality problems through planned and systematic activities including documentation.
Involves a series of Inspections and reviews to ensure that product meets its specifications.	It is a complete system to assure the quality of products or services; involves audits, reviews, and proper testing tools to meet customer satisfaction.

Cost of Quality – A financial measure:

- Cost of quality is the total amount the company spends to achieve and survive with the quality of its product.
- Quality costs are divided into two categories:
  1. Costs due to poor quality (Failure costs).
  2. Costs related to improvement in quality (Prevention and Appraisal costs).
- Quality costs are divided into costs associated with Prevention, Appraisal, and Failure.
  - **Prevention costs:** It includes quality planning, formal technical reviews, test equipment, staff training, and clear specification. i.e. Cost of preventing customer dissatisfaction, including errors or weaknesses in software, design, documentation, and support.

- **Appraisal costs:** It includes equipment maintenance, testing, in-process and inter-process inspection. i.e. Cost of inspection and testing.
- **Failure costs:** Failure costs would disappear if no defects appeared before shipping a product to customers. Internal and external failure costs are the two types of failure costs.

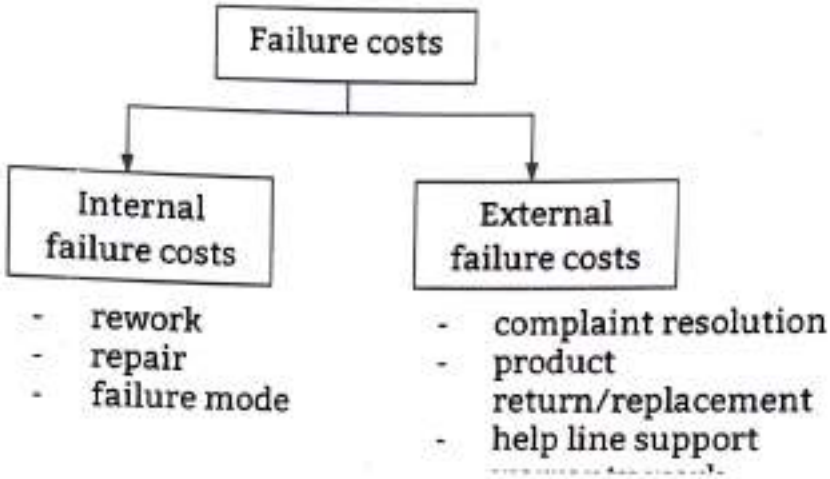


Fig. 6.4: Types of Failure Costs

1. **Internal failure costs:** Includes rework, repair, and failure mode analysis.
2. **External failure costs:** Includes complaint resolution, product return and replacement, helpline support, and warranty work. External failure costs are associated with the defects found after the shipment of the product to the customer.

6.4.4 Quality Movement

- The quality movement began in the 1940s with the seminal work of W. Edwards Deming and had its first true test in Japan.
- A quality movement has begun to enable organizations to produce high-quality deliverables. This movement has gone through four steps:
  1. The first stage in the Quality Movement is to create a visible, repeatable, and measurable process (in this case, the software process).
  2. The second step investigates the invisible that affect the process and works to minimize their impact. For example, high staff turnover, which is caused by constant reorganization within a company, may have an impact on the software process. Perhaps a stable organizational structure could significantly improve software quality.
  3. While the first two steps focus on the process, the next step concentrates on the user of the product (in this case, software). In essence, by examining the way the user applies the product, this step leads to improvement in the product itself and, potentially, to the process that created it. It examines the way the product is used by the customer to improve both the product and the development process.

4. The last step expands considerations beyond the immediate product. This is a business-oriented step in which product usage in the marketplace is observed to discover new product applications and identify new products to develop.

6.4.5 Background Issues

- Software Quality Assurance is an umbrella activity that is applied throughout the software process. Quality control and assurance are essential activities for the business that produces products to be used by others. Many constituencies have software quality responsibility: software engineers, project managers, customers, salespeople, and the individuals who serve within the SQA group. The people who perform SQA must look at the software from the customer's point of view.

6.4.6 SQA Activities

- Software Quality Assurance (SQA) is the function of software quality that assures that the standards, processes, and procedures are appropriate for the project and are correctly implemented.
- SQA assures that standards and procedures are established and are also followed throughout the software life cycle.
- SQA activities are planned. The Software Engineering Institute (SEI) recommends a set of SQA activities that address quality assurance planning, record keeping, analysis, and reporting.
- The following are some of the SQA activities carried out by an independent SQA group:

1. **Preparation of SQA plan for the project.**
   - The plan identifies:
     - Evaluations to be performed.
     - Audits and reviews to be performed.
     - Standards applicable to the project.
     - Error tracking and reporting procedures.
     - Documents to be produced by the SQA group.
2. **Participate in the development of the project's software process description.**
   - The software team selects a work process to be performed. The SQA group examines the process description to ensure that it complies with organizational policy, internal and external software standards.
3. **Review software engineering activities (Analysis, Design, Construction, Verification, and Management) to verify compliance with the defined software process.**
   - The SQA team detects, reports, and tracks process irregularities, as well as ensures that corrections have been made.

4. **Audit designated software work products**
   - Audit designated software work products to verify compliance with those defined as part of the software process.
   - SQA group also verifies that corrections are done and reported to the project manager.
5. **Ensure that any deviations in software or work products are documented and handled according to a documented procedure.**
   - Deviations may be encountered in the project plan, process description, applicable standards, or technical work products.
6. **Record any evidence of non-compliance and report them to management.**
   - Noncompliance items are tracked until they are resolved.
7. **Configuration Management Monitoring:**
   - Configuration Management is also called Change Control Management. It is a systematic way of controlling software changes. It also helps to store and retrieve the configurable items (i.e. Software Code, Defect log, Test Reports, etc.).
   - SQA ensures that SCM activities are carried out following Configuration Management plans, methods, and standards. SQA examines Configuration Management plans for conformity to SCM policies, requirements and follows up on non-conformances.

Configuration Management activities:

- The Configuration Management activities are monitored and audited by SQA, it includes:

1. **Baseline control:**
   - Baselining is done when the product is ready to go to the next phase.
   - The current state of the product is preserved and further changes to the product are done along with the change in version of the product.
2. **Configuration identification:**
   - Software configuration identification is constant and accurate for the naming of programs, software modules, and their associated software documents.
   - Example tools for configuration management – Win CVS (Concurrent Version System), VSS (Visual Source Safe).
   - SQA assures Verification and Validation by monitoring technical reviews, inspections, and walkthroughs.

6.5 FORMAL APPROACHES TO SQA

- Formal approaches of SQA are Proof of Correctness, Statistical Quality Assurance, and Cleanroom Process.

6.5.1 Proving Programs/Specifications are Correct

- Logically prove that requirements have been correctly transformed into programs (e.g., prove assertions about programs).
- A proof of correctness is a mathematical proof that a computer program or a part thereof will, when executed, return correct results i.e., results fulfilling specific requirements. Before proving a program correct, the theorem to must be proved and formulated.

6.5.2 Statistical Quality Assurance

- Statistical quality assurance involves the following steps:

1. **Information about software defects is collected and categorized:**
   - Software defects are gathered and classified as Low, Medium, and Complex, or Serious, Moderate, and Low.
2. **Each defect is traced back to its cause:**
   - Each problem is traced back to its root cause, such as non-conformance to requirements, design fault, standard violation, poor customer communication, and so on.
3. **Use of Pareto principle:**
   - The Pareto principle(as the 80/20 rule) is a theory maintaining that 80 percent of the output from a given situation or system is determined by 20 percent of the input.
   - Using the Pareto principle (80% of the defects can be traced to 20% of the causes) isolate the 20 % ("vital few" defect causes). Once the vital few causes have been identified, move to correct the problems that caused the defects.
4. **Move to correct the problems that caused the defects:**
   - On a parts-per-million (PPM) basis, SQA is used to discover potential differences in the manufacturing process and predict potential faults.
   - It gives a statistical description of the finished product and addresses quality and safety concerns that may develop during production.

6.5.3 Cleanroom Process

- The objective of the Cleanroom methodology is to achieve or approach zero defects with specialized reliability. It provides a complete discipline within which software personnel can plan, specify, design, verify code, test and certify the software.
- The name Cleanroom was taken from the electronics industry, where a physical cleanroom exists to prevent the introduction of defects during hardware fabrication.
- The Cleanroom process is built upon an incremental development plan (stepwise
- refinement) that provides a basis for statistical quality control of the development process.
- Each increment is a complete iteration of the process, and measures of performance in each iteration are compared with pre-established standards to determine whether or not the process is "in control."

- If quality standards are not met, developers return to the design stage without testing the further increments. The feedback produced in each increment is used for project management and process improvement. Fig. 6.5 shows the full implementation of the Cleanroom process.
- In a Cleanroom development, correctness verification replaces unit testing and debugging. After coding is complete, the software immediately enters the system test with no debugging.

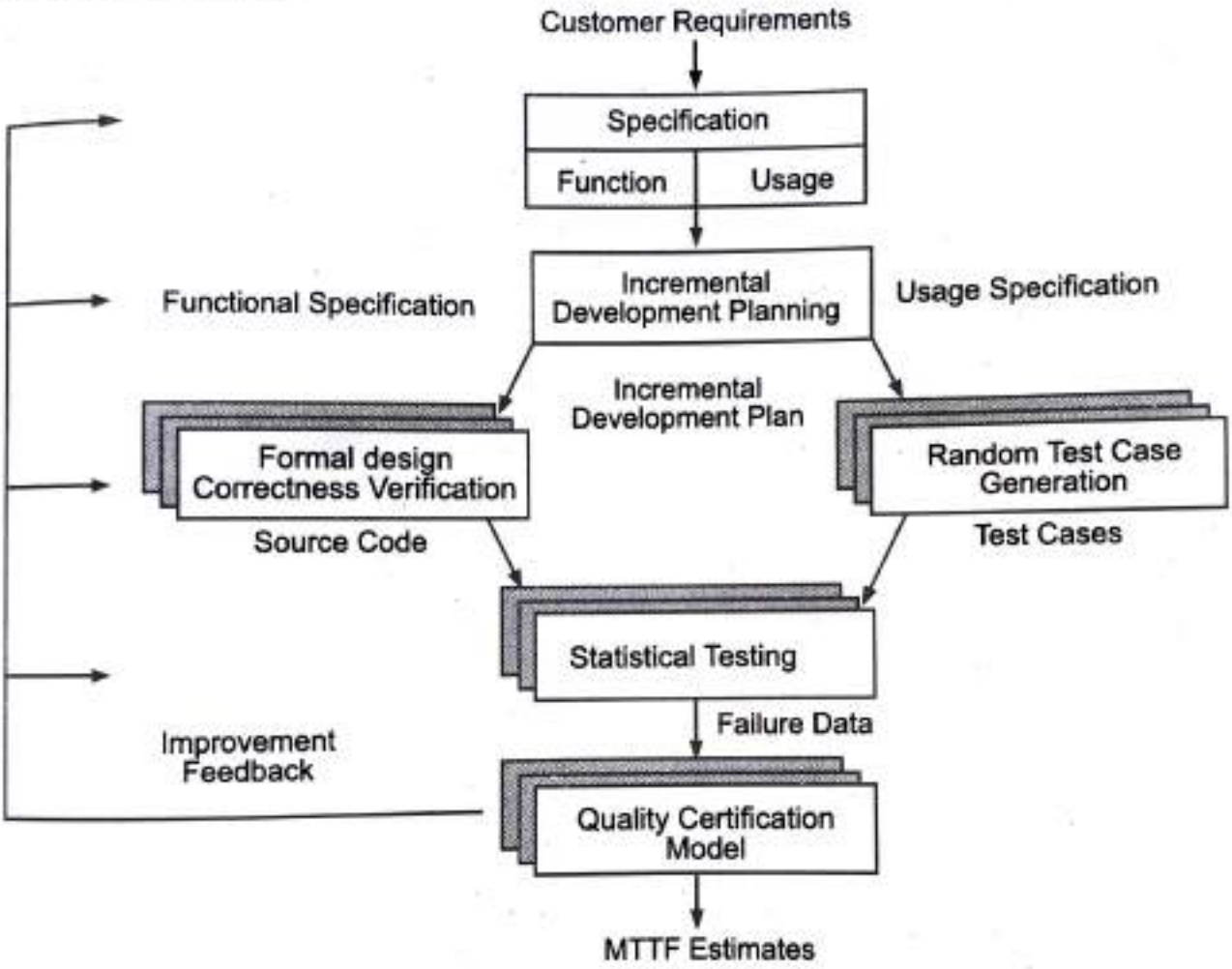


Fig. 6.5: Cleanroom process

- Cleanroom specifications are built upon box structure specifications and design. Box structure specifications begin with a black-box specification in which the expected behavior of the system is specified in terms of the system responses and transition rules.
- Black boxes are then translated into state boxes that define encapsulated state data required to satisfy black-box behavior.
- Finally, white box designs are created, which define the procedural design of services based on state data to satisfy black-box behavior.
- Small teams, within a team for large projects, carry out Cleanroom operations. The project team works in a disciplined fashion to ensure the intellectual control of work.

Team reviews are performed to verify the correctness of every condition in the specification.

- Proof of correctness and formal quality certification by statistical testing are the foundations of the Cleanroom procedure. According to the procedure, statistical testing might take the position of coverage and path testing. All testing in Cleanroom is based on a predictable client usage pattern.
- The purpose of test cases is to practice the most commonly used functions. As a result, faults that are likely to cause frequent failures for users will be discovered first.
- As a result, Cleanroom Software Engineering produces software that is mathematically good in design and statistically valid in testing.

6.6 SOFTWARE RELIABILITY

- Software reliability is the probability of failure-free operation of a computer program for a specified period in a specified environment. Reliability is a customer-oriented view of software quality.

Software Reliability Metrics:

- Software reliability metrics act as a benchmark to quantitatively express the reliability of software products.
- Given below are some of the metrics that could be followed to express trust and dependability of software products:
  - Mean Time to Failure *or* MTTF (average time between non-repairable failures)
  - Mean Time to Repair *or* MTTR (average time required to troubleshoot and repair failed equipment)
  - Mean Time Between Failure or MTBF.

    MTBF = MTTF + MTTR
  - Rate of Occurrence of Failure or ROCOF.
  - Probability of Failure on Demand or POFOD.
  - Availability

Software Availability:

- The probability that a program is operating according to requirements at a given point in time and is defined as Availability.

  Availability = [MTTF/(MTTF + MTTR)] x 100%

- MTBF reliability measure is equally sensitive to MTTF and MTTR.
- Availability measure is more sensitive to MTTR, an indirect measure of the maintainability of software.

6.7 THE ISO 9000 QUALITY STANDARDS

- ISO 9000 is defined as a set of international standards on quality management and quality assurance developed to help companies effectively document the quality system elements needed to maintain an efficient quality system.

- The types of industries to which the various ISO standards apply are as follows.
  1. **ISO 9001:** This standard applies to the organizations engaged in the design, development, production, and servicing of goods. This is the standard that applies to most software development organizations.
  2. **ISO 9002:** This standard applies to those organizations which do not design products but are only involved in the production. Examples of these category industries contain steel and car manufacturing industries that buy the product and plants designs from external sources and are engaged in only manufacturing those products. Therefore, ISO 9002 does not apply to software development organizations.
  3. **ISO 9003:** This standard applies to organizations that are involved only in the installation and testing of the products. For example, Gas companies.

How to get ISO 9000 Certification?

- An organization that determines to obtain ISO 9000 certification applies to the ISO registrar's office for registration. The process consists of the following stages:
  1. **Application:** When a company decides to pursue ISO certification, it must apply for registration with the registrar.
  2. **Pre-Assessment:** During this stage, the registrar makes a rough assessment of the organization.
  3. **Document review and Adequacy of Audit:** During this stage, the registrar examines the organization's document and makes suggestions for improvement.
  4. **Compliance Audit:** The registrar evaluates whether the organization has complied with the suggestions made during the review at this step.
  5. **Registration:** The Registrar awards the ISO certification after the successful completion of all the phases. Registration is simply the audit and approval of your quality system against ISO standards by an independent registered auditor
  6. **Continued Inspection:** The registrar continued to monitor the organization time by time.

6.8 SQA PLAN

- Planning is one of the most important aspects of SQA. The entire operation of the SQA team depends on how well their planning is done.
- SQA plan provides a road map for establishing software quality assurance.
- The plan serves as a template for various SQA activities that are instituted for each software project.
- It defines the techniques, procedures, and methodologies that will be used to assure timely delivery of the software that meets specified requirements within project resources.
- SQA Plan helps everyone involved in the project to know in advance how the application will work in the actual environment.

- IEEE [IEE94] has published a standard for SQA plans. A structured SQA plan identifies:

1. **The purpose and scope of the plan.**
   - This section list the software covered by the SQA plan.
   - Also, the state portion of the software life cycle is covered.
2. **A description of all software engineering work products.**
   - Models, documents (software requirements and design specification, user manual), and source code.
3. **All applicable standards and guidelines are applied during the software life cycle.**
   - Design, Coding, and testing standards.
   - ISO 9001:2000 – A quality assurance standard that is applied to software engineering. It is a special set of rules which are meant to be followed by an organization to understand customer needs and expectations. ISO defines the formal quality management system to enhance customer satisfaction.
   - Six Sigma.
4. **SQA actions and tasks.**
   - Defines what reviews or audits are to be done.
   - When and how they will be done?
   - What further actions are required?
   - For example, Software requirements reviews, Design review, Final software, and documentation review.
5. **Tools and methods supporting SQA actions and tasks.**
   - Formal Technical Review.
   - Review reporting and record keeping.
   - Review guidelines.
6. **Software Configuration Management (SCM) Procedures for managing change.**
7. **Methods for assembling, preserving, and maintaining all SQA-related records.** i.e. securing SQA documents.
8. **Organizational roles and responsibilities are relative to product quality.**

6.9 SIX SIGMA

- Sigma (σ the Greek symbol) is a term used in statistics that measure standard deviation.
- In business, it is an indication of process defects and how far these defects deviate from perfection.
- The term "Six σ" was suggested by Bell-Smith, an Engineer with Motorola.
- Six Sigma represents a stringent level of quality. It is the measure of quality that strives for near perfection.

- It is a disciplined, data-driven method for eliminating defects. According to Six Sigma; a defect is defined as anything that falls outside the customer's specifications.
- It's a method of creating goals for reducing service problems, which are linked to customer needs and satisfaction.

6.9.1 Normal Distribution Curve

- Six Sigma is a statistical concept which measures a process in terms of defects. At the Six Sigma level, there are only 3.4 defects per million opportunities.
- It is also a philosophy of eliminating defects through practices that highlight understanding, measuring, and improving processes.
- Fig. 6.6 indicates the areas under the curve of normal distribution defined by Standard Deviation in terms of percentage.
- The area under the curve as defined by plus/minus one standard deviation (sigma) from the mean is 68.26%. Area defined by:

plus/minus 2 Standard Deviation is	:	95.44%
plus/minus 3 Standard Deviation is	:	99.73%
plus/minus 4 Standard Deviation is	:	99.9937%
plus/minus 5 Standard Deviation is	:	99.999943%
plus/minus 6 Standard Deviation is	:	99.9999998%

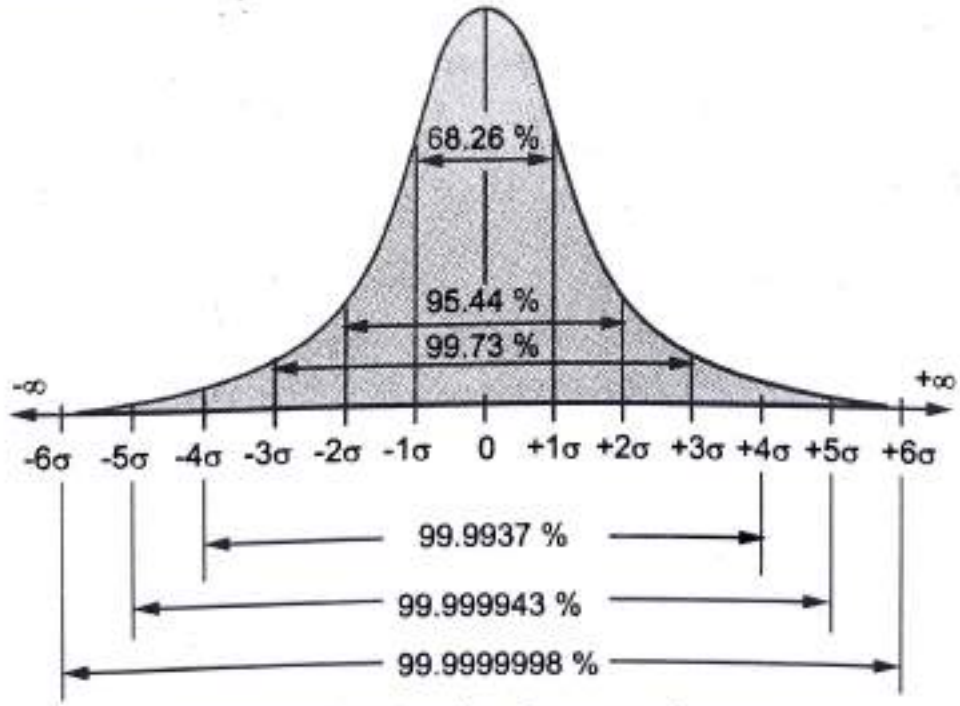


Fig. 6.6: Areas under the normal curve

- Area outside the 6 Sigma area is thus 100% - 99.9999998% = 0.0000002%
- If the area within the Six Sigma limit is considered as a percentage of DEFECT-FREE parts and the area outside the limit as a percentage of DEFECTIVE parts, then Six Sigma is equal to 2 defectives per billion parts or 0.002 defective parts per million (ppm).

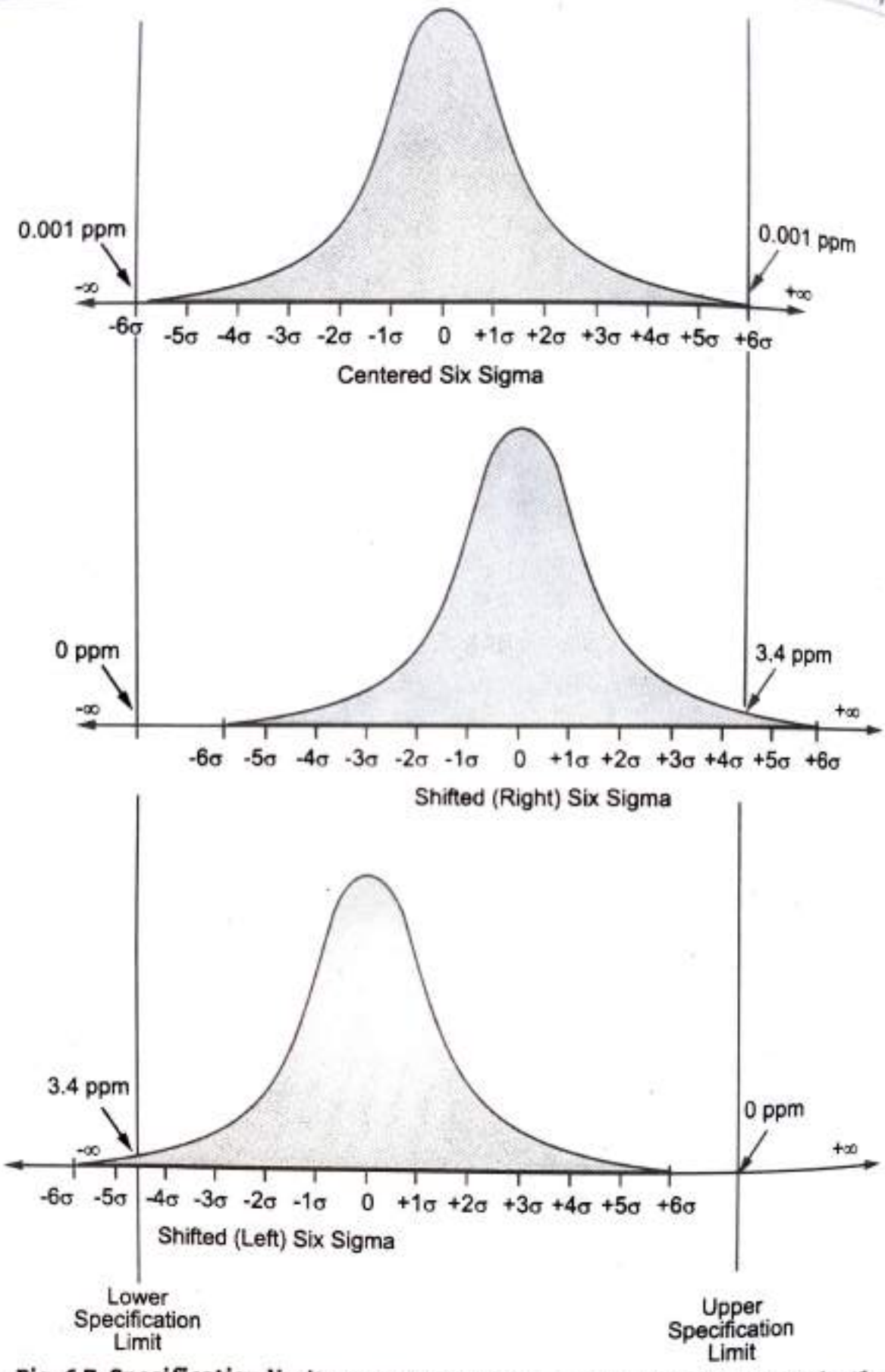


Fig. 6.7: Specification limits, centered Six Sigma and shifted (1.5) Six Sigma

- Given the specification of customers' requirements, the purpose is to produce software within the limits. Products outside the specification limits do not meet the requirements. 6 Sigma quality level is said to be maintained; if and only if sigma variations are minimized to specification limits by changing the development process.

- The sigma level of a process is where its customer-driven specifications intersect with its distribution. A centered Six Sigma process as shown in Fig. 6.7 has a normal distribution with mean = target and specifications are placed 6 standard deviations to either side of the mean.
- At this point, the portions of the distribution beyond the specification limits contain 0.002 ppm of the data (0.001 on each side). It is assumed that each execution version of the process will produce the exact distribution. But in reality, most manufacturing processes experience a shift of 1.5 standard deviations so that the mean no longer equals the target. Because of this, a larger portion of the distribution extends beyond the specification limits i.e. 3.4 ppm. Such shifting is illustrated in the two lower panels of Fig. 6.7.
- Given fixed specification limits, the distribution of the production process may shift either to left or right. When the shift is 1.5 sigma, the area outside the specification limit on one end is 3.4 ppm, and on the other, it is nearly zero.

Six Sigma is distinguished from other quality systems by several themes:

- Focus on the customer rather than process, inputs, or outputs. The sigma level is determined by how well the organization and its processes meet customer requirements.
- Data and fact-driven management emphasize on measurement of quantitative data.
- The process is the key factor of success. Six Sigma tools are applied to create major process changes and incremental improvements, to improve the process quality.
- Emphasis on root causes, digging down beyond proximal causes to find what is going on.
- Continual change is necessary to guarantee that these changes are maintained over time.

6.9.2 Methodology of Six Sigma Design

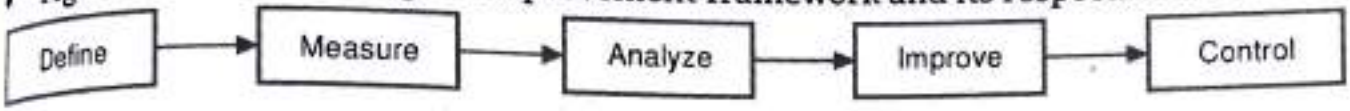
- The primary goal of Six Sigma is to improve customer satisfaction by reducing and eliminating defects.
- Defects may be related to any aspect of customer satisfaction: product quality, scheduled delivery, or minimum cost.
- The Six Sigma journey of defect reduction, process improvement, and customer satisfaction is based on the "statistical thinking" paradigm, where:
  - Everything is a process.
  - All processes have inherent variability.
  - Data is used to understand the variability and drive process improvement decisions.
- Key steps in the Six Sigma improvement framework for implementing the statistical thinking paradigm are:
  1. Define - Measure - Analyze - Improve - Control (DMAIC).
  2. Define - Measure - Analyze - Design - Verify (DMADV).

- DMAIC and DMADV are two Six Sigma methodologies that remove defects from a process or product.
- The **DMAIC Methodology** is applied when a product or process exists but does not satisfy client requirements or performs poorly.

D: Define	**Identify and state the practical problem.** 1. Initiate the project. 2. Prepare a project charter (including problem statement, goals, constraints or assumptions, scope, rules, and a preliminary plan). 3. Recognize the client and transform the "voice of the client" into requirements. 4. Create a high-level process diagram/design.
M: Measure	**Validate the problem through rigorous data collection.** 1. Gather data on outputs, outcomes, processes, and inputs. 2. Discover and prioritize customer needs. 3. Identify facts and data that offer clues to quality issues. 4. Create an early sigma measure of the process i.e. measure baseline performance.
A: Analyze	**Convert the practical problem to a statistical one, Define the statistical goal and Identify a potential statistical solution.** 1. Analyze the data, using advanced statistics and tools as needed. 2. Find the root cause of quality issues. 3. Develop design alternatives.
I: Improve	**Confirm and Test the statistical solution** 1. Solution and action stage: Solve the problem and act on it. 2. It is possible to return to the Charter and revise the problem/goal statement in light of new insights. 3. If necessary, confirm with management. 4. If necessary, change the project's scope. 5. Implement, manage and test the solutions. Solutions are thoroughly reviewed and tested before full implementation.

C: Control	**Convert the statistical solution to a practical solution.** 1. Develop and implement a monitoring process to track changes and results. 2. Create a response plan in case solutions do not work as intended. 3. Help management focus on appropriate metrics to get info on outcomes and processes. 4. Handoff responsibilities to day-to-day operations staff. 5. Ensure management support for long-term goals.

- Fig. 6.8 shows the Six Sigma improvement framework and its respective toolkit.



• Benchmark • Baseline • Charter • Voice of customer • Project management • Quality function deployment	• Basic tools • Defects metrics • Data collection • Sampling techniques	• Cause and effect diagram • Control charts • Reliability analysis • Root cause analysis	• Design of experiments • Modelling • Tolerance • Robust design	**Statistical controls :** • Control charts • Time series methods **Non-statistical controls:** • Performance management • Preventive activities

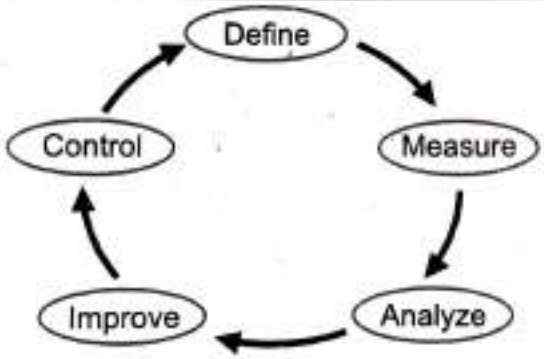
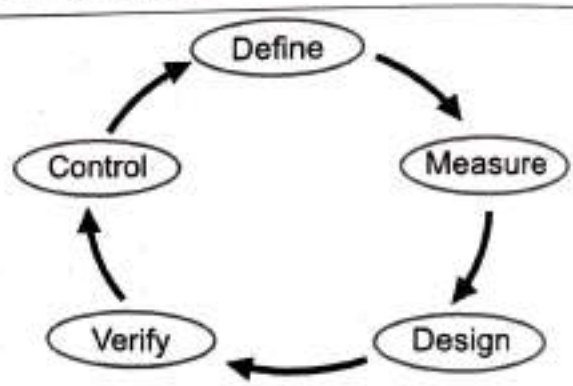
Fig. 6.8: Six Sigma Improvement Framework and Toolkit

- The **DMADV methodology** is used when:
  - A product or process does not exist at the company and one needs to be developed.
  - Product or process does exist and has been optimized (using either DMAIC or not) and still doesn't meet the level of customer specification or Six Sigma level.

D: Define	• Determine the project to be undertaken, its objective, and scope.
M: Measure	• Determine the 'Voice Of the Customer (VOC)'. • Convert it into 'Critical to Quality (CTQ)', obtain data to quantify. • Determine process performances.
A: Analyze	• Analyze data to develop design concepts and produce a high-level design.
D: Design	• Develop detail design. • Validate and implement the new design. • Integrate with the existing system.
V: Verify	Check the completed design and ensure its transition to the customer.

6.9.3 The Difference between DMAIC and DMADV Methodologies

Table 6.2: DMAIC Vs DMADV

Sr. No.	DMAIC	DMADV
1.	Define the project goals and customer (internal and external) deliverables.	Define the project goals and customer (internal and external) deliverables.
2.	Measure the process to determine current performance.	Measure and determine customer needs and specifications.
3.	Analyze and determine the root cause(s) of the defects.	Analyze the process options to meet the customer needs.
4.	Improve the process by eliminating defects.	Design (detailed) the process to meet the customer's needs.
5.	Control future process performance.	Verify the design performance and ability to meet customer needs.
6.	 Fig. 6.9(a)	 Fig. 6.9(b)

6.10 INFORMAL REVIEWS

- An informal review is an informal static test technique performed on any kind of requirements, design, code, or project plan. During the informal review, the work product is given to a domain expert and the feedback/comments are reviewed by the owner/author.
- It is also called peer-review i.e. simply giving a document to a colleague and asking them to look at it closely which will identify defects we might never find on our own.
- No agenda is required and the results are also not formally reported.
- An informal review is a review that is not based on a documented procedure, i.e. formal procedure. This can be applied at various times during the early stage of development.
- These reviews are conducted between the two-person team. In later stages, more people are involved.
- Informal reviews aim to improve the quality of the document and help the authors.

Summary

- Software testing tools are applications that can be used to assist developers and testers in performing manual or automated tests.
- JUnit is an open-source Unit Testing framework for Java. It is useful for Java Developers to write and run repeatable tests.
- Apache JMeter is a testing tool used for analyzing and measuring the performance of different software services and products.
- WinRunner is a widely used Automated Software Testing Tool for Functional testing.
- LoadRunner is used to test the Client/Server applications such as database management systems and websites.
- Rational Robot is an automated testing tool for functional and regression testing for automating Windows, Java, IE, and ERP applications that run on the Windows platform.
- Software Quality Assurance (SQA) is a process that assures that all software engineering processes, methods, activities, and work items are monitored and comply with the defined standards.
- Software Reliability is the ability of a system or component to perform its required functions under static conditions for a specific period.
- The Software Quality Assurance Plan (SQAP) is a comprehensive plan that directs the work of the SQA function for a year.
- ISO 9000 is defined as a set of international standards on quality management and quality assurance developed to help companies effectively document the quality system elements needed to maintain an efficient quality system.
- Six Sigma is a method that provides organizations tools to improve the capability of their business processes.
- An informal review is an informal static test technique performed on any kind of requirements, design, code, or project plan.

Check Your Understanding

1. Which one of the following is a Software Testing Tool?
   (a) Rational Rose (b) Oracle
   (c) WinRunner (d) Linux
2. Quality Management in software engineering is also known as ____.
   (a) SQA (b) SQM
   (c) SQI (d) SQA and SQM
3. What is Six Sigma?
   (a) It is the most widely used strategy for statistical quality assurance.
   (b) It refers to six standard deviations.
   (c) It is the most widely used strategy for statistical quality assurance AND the "Six Sigma" refers to six standard deviations.
   (d) A Formal Technical Review (FTR) guideline for quality walkthrough or inspection.

4. According to ISO 9001, inspection and testing come under which management responsibility?
   (a) Process control.
   (b) Document control.
   (c) Control of nonconforming products.
   (d) Servicing.
5. The probability of failure-free operation of a software application in a specified environment for a specified time.
   (a) Software Reliability (b) Software Quality
   (c) Software Availability (d) Software Safety
6. The main purpose of the analyze phase is to _____.
   (a) Identify possible solutions (b) Create a pilot plan
   (c) Identify and validate root causes (d) All of the above
7. Simple graphical displays are used in the Measure phase to _____.
   (a) Show baseline information (b) Determine stability
   (c) Represent central tendency (d) All of the above
8. An informal review may consist of_____.
   (a) Casual meeting (b) Correction
   (c) Inspection (d) Pair programming
9. The ISO 9000 series was created by the _____.
   (a) Internal Organization for Standardization
   (b) International Trade Organization.
   (c) International Organization for Standardization.
   (d) International Standards for Organization.
10. Statistical quality assurance involves _____.
   (a) using sampling in place of exhaustive testing of software.
   (b) surveying customers to find out their opinions about product quality.
   (c) tracing each defect to its underlying cause, isolating the "vital few" causes, and moving to correct them
   (d) tracing each defect to its underlying causes and using the Pareto principle to correct each problem found.

Answers

1. (c)	2. (a)	3. (c)	4. (a)	5. (a)
6. (c)	7. (d)	8. (a)	9. (a)	10. (c)

Practice Questions

Q.I Answer the following questions in short.

1. What is a test case design?
2. Which are the techniques of test case design?
3. What is an Apache JMeter?
4. What is quality?
5. Define the term SQA.
6. What is the WinRunner testing tool?

Q.II Answer the following questions.

1. Explain the test case design for the ATM system.
2. Explain in detail the JUnit testing framework.
3. Which are the features of the 'Rational Robot' test tool?
4. Explain test case design for the login process.
5. Explain the test case design format?
6. Explain the various ways to measure reliability.

Q.III Define the terms.

1. Apache JMeter
2. LoadRunner
3. Vusers
4. Scenario

★★★

Solved Question Paper
Summer 2022

Time : 2 ½ Hours | Marks : 70

Q.1 Attempt any EIGHT of the following. (Out of TEN) [8 × 2 = 16]

(a) Explain terms - Error, Fault and Failure.

Ans. Refer to Section 1.1.1.

(b) Define software testing.

Ans. Refer to Section 1.1.

(c) What is structural testing?

Ans. Refer to Section 2.2.

(d) How to calculating cyclomatic complexity?

Ans. Refer to Section 4.3.

(e) What is verification testing?

Ans. Refer to Section 3.7.2.

(f) Explain types of Acceptance testing?

Ans. Refer to Section 3.6.2.

(g) Define software metrics?

Ans. Refer to Section 4.2.1.

(h) What is user documentation testing?

Ans. Refer to Section 5.4.1.

(i) Define the term SQA.

Ans. Refer to Section 6.1.

(j) What is a test case design?

Ans. Refer to Section 2.3.

Q.2 Attempt any FOUR of the following (Out of FIVE) : [4 × 4 = 16]

(a) What is debugging? Explain with its phases.

Ans. Refer to Section 1.7.2.

(b) Explain in details verification and validation.

Ans. Refer to Section 3.7.

(c) What is Black - Box testing? Explain with its techniques.

Ans. Refer to Sections 2.4 and 2.4.1.

(d) Write difference between static testing and Dynamic testing.

Ans. Refer to Section 1.4.3.

(e) Explain GUI testing in details.

Ans. Refer to Section 5.2.

Q.3 Attempt any FOUR of the following (Out of FIVE) : [4 × 4 = 16]

(a) What is difference between client/server testing and web - based testing?

Ans. Refer to table 5.1, chapter 5.

(b) Explain five different level of capability maturity model (CMM).

Ans. Refer to Section 4.2.2.

(c) Explain Acceptance testing in details.

Ans. Refer to Section 3.6.

(d) Explain Top - Down and Bottom - UP integration testing in details.
Ans. Refer to Sections 3.4.1 and 3.4.2.
(e) Explain term unit testing.
Ans. Refer to Section 3.3.

Q.4 Attempt any FOUR of the following (Out of FIVE) : **[4 × 4 = 16]**

(a) Explain testing principles in details.
Ans. Refer to Section 1.3.
(b) Explain Load testing and stress testing in details.
Ans. Refer to Section 3.10.2.
(c) Write difference between Quality Assurance (QA) and Quality control (QC).
Ans. Refer to table 6.1 of chapter 6.
(d) Explain test case design for login process.
Ans. Refer to Section 2.3.
(e) Explain software testing life cycle (STLC) in details.
Ans. Refer to Section 1.7.

Q.5 Write a short note on Any TWO of the following (Out of THREE) : **[2 × 3 = 6]**

(a) Load Runner.
Ans. Refer to Section 6.3.1.
(b) Testing for Real - Time system
Ans. Refer to Section 5.5.
(c) Goal - Question - Metric Model (GQM).
Ans. Refer to Section 4.2.4.

Winter 2022

Time : 2 ½ Hours **Marks : 70**

Q.1 Attempt any EIGHT of the following. (Out of TEN) **[8 × 2 = 16]**

(a) What is Software Testing?
Ans. Refer to Section 1.1.
(b) What is Static testing?
Ans. Refer to Section 2.1.
(c) State the advantages of manual testing.
Ans. Refer to Section 3.11.1.
(d) What are formulae for calculating cyclomatic complexity?
Ans. Refer to Section 4.3.
(e) What is Gray-box testing?
Ans. Refer to Section 2.5.
(f) Define validation testing?
Ans. Refer to Section 3.7.1.
(g) What is Debugging?
Ans. Refer to Section 1.7.2.
(h) Explain terms - Error, Fault and Failure?
Ans. Refer to Section 1.1.1.

(i) Define regression testing.

Ans. Refer to Section 3.11.

(j) What is software metric?

Ans. Refer to Section 4.2.1.

Q.2 Attempt any FOUR of the following. (Out of FIVE) **[4 × 4 = 16**

(a) Write difference between verification and validation.

Ans. Refer to table 1.1 of chapter 1.

(b) Explain software testing life cycle with diagram.

Ans. Refer to Section 1.7.

(c) Explain Boundary - Value analysis in details.

Ans. Refer to Section 2.4.1.

(d) Explain Acceptance testing in details.

Ans. Refer to Section 3.6.

(e) Explain Test Case Design along with example.

Ans. Refer to Section 2.3.

Q.3 Attempt any Four of the following. (Out of Five) **[4 × 4 = 16**

(a) Explain any four testing principles in detail.

Ans. Refer to Section 1.3.

(b) Explain white box testing and its techniques.

Ans. Refer to Section 2.2.

(c) Explain Sandwich and Big-Bang approach of Integration testing.

Ans. Refer to Sections 3.8 and 3.9.

(d) Explain load and Smoke testing in detail.

Ans. Refer to Sections 3.12 and 3.13.

(e) Write difference between Static and Dynamic testing.

Ans. Refer to table 2.1 of chapter 2.

Q.4 Attempt any Four of the following. (Out of Five) **[4 × 4 = 16]**

(a) Explain test case design for the login process.

Ans. Refer to Section 2.3.

(b) Stub and Driver concept in Unit testing.

Ans. Refer to Section 3.3.

(c) Explain GUI testing in details.

Ans. Refer to Section 5.2.

(d) What is difference between client/server and web-based testing?

Ans. Refer to Section table 5.1 of chapter 5.

(e) Calculate the cyclometric complexity of a code which accepts 3 integer values and print the highest and lowest value.

Ans. Refer to example 1 of chapter 4.

Q.5 Write a short note on any Two of the following. (Out of Three) **[2 × 3 = 6]**

(a) Testing for Real - Time System.

Ans. Refer to Section 5.5.

(b) Stub and Driver concept in unit testing.

Ans. Refer to Section 3.3.

(c) Load Runner.

Ans. Refer to Section 6.3.1.

Summer 2023

Time : 2 ½ Hours **Marks : 70**

Q.1 Attempt any EIGHT of the following. (Out of TEN) **[8 × 2 = 16]**

(a) What is Software Testing?

Ans. Refer to Section 1.1.

(b) What is Static testing?

Ans. Refer to Section 2.1.

(c) State the advantages of manual testing.

Ans. Refer to Section 3.11.1.

(d) What are formulae for calculating cyclomatic complexity?

Ans. Refer to Section 4.3.

(e) What is Gray-box testing?

Ans. Refer to Section 2.5.

(f) Define validation testing?

Ans. Refer to Section 3.7.1.

(g) What is Debugging?

Ans. Refer to Section 1.7.2.

(h) Explain terms - Error, Fault and Failure?

Ans. Refer to Section 1.1.1.

(i) Define regression testing.

Ans. Refer to Section 3.11.

(j) What is software metric?

Ans. Refer to Section 4.2.1.

Q.2 Attempt any FOUR of the following. (Out of FIVE) **[4 × 4 = 16]**

(a) Write difference between verification and validation.

Ans. Refer to table 1.1 of chapter 1.

(b) Explain software testing life cycle with diagram.

Ans. Refer to Section 1.7.

(c) Explain Boundary - Value analysis in details.

Ans. Refer to Section 2.4.1.

(d) Explain Acceptance testing in details.

Ans. Refer to Section 3.6.

(e) Explain Test Case Design along with example.

Ans. Refer to Section 2.3.

Q.3 Attempt any Four of the following. (Out of Five) [4 × 4 = 1

(a) Explain any four testing principles in detail.

Ans. Refer to Section 1.3.

(b) Explain white box testing and its techniques.

Ans. Refer to Section 2.2.

(c) Explain Sandwich and Big-Bang approach of Integration testing.

Ans. Refer to Sections 3.8 and 3.9.

(d) Explain load and Smoke testing in detail.

Ans. Refer to Sections 3.12 and 3.13.

(e) Write difference between Static and Dynamic testing.

Ans. Refer to table 2.1 of chapter 2.

Q.4 Attempt any Four of the following. (Out of Five) [4 × 4 = 1

(a) Explain test case design for the login process.

Ans. Refer to Section 2.3.

(b) Stub and Driver concept in Unit testing.

Ans. Refer to Section 3.3.

(c) Explain GUI testing in details.

Ans. Refer to Section 5.2.

(d) What is difference between client/server and web-based testing?

Ans. Refer to table 5.1 of chapter 5.

(e) How to calculate the cyclometric complexity of a code? Explain with example.

Ans. Refer to example 1 of chapter 4.

Q.5 Write a short note on any Two of the following. (Out of Three) [2 × 3 =

(a) Testing for Real - Time System.

Ans. Refer to Section 5.5.

(b) Stub and Driver concept in unit testing.

Ans. Refer to Section 3.3.

(c) Load Runner.

Ans. Refer to Section 6.3.1.

★★

OUR MOST RECOMMENDED
TEXT BOOKS FOR T.Y. B.B.A. : SEMESTER-VI

- Essentials of E-Commerce : Dr. Gautam Bapat
- Management Information System : Jayant Oke
- Business Project Management : Dr. Sheetal Randhir, Dr. Kabeer Kharade
- Management of Innovations and Sustainability : Dr. Leena Modi (Gandhi)
- International Brand Management : Dr. Shaila Bootwala, Dr. Sadia Merchant
- Cases in Marketing Management + Project : Dr. Shaila Bootwala, Uzma Shaikh, Dr. Raisa Shaikh, Dr. Farzana Shaikh
- Financial Management : Archana Vechalekar, Dr. Smita Wadaskar, Mrs. Rekha Kankariya
- Cases in Finance + Project : Dr. Ameya Anil Patil, Dr. Swami
- Global Human Resource Management : Dr. Leena Modi (Gandhi), Dr. Shalaka Parker, Mrs. Viral Ahire
- Recent Trends and HR Accounting + Project : Ankita Bhatt, Karan Randive, Dr. Anamika Ghosh

OUR MOST RECOMMENDED
TEXT BOOKS FOR T.Y. B.B.A. (I.B.) : SEMESTER-VI

- New Venture Creation & Start-Ups : Arati Oturkar, Dr. Jyoti Joshi
- International Project Management : Dr. Devangi Deore, Dr. Nutan Thoke
- Decision Making and Risk Management : Dr. Ameya Anil Patil
- Management of Agribusiness and Agri Export : Dr. Roopali Sheth, Asmita Kulkarni, Dr. Nutan Thoke
- International Service Management : Diya Tanmay Devare
- International HRM : Rutuja Purohit, Pasmina Doshi

OUR MOST RECOMMENDED
TEXT BOOKS FOR T.Y. B.B.A. (C.A) : SEMESTER-VI

- Recent Trends in IT : Dr. Pallawi Bulakh, Meenal Jabde
- Software Testing : Dr. Aruna A. Deoskar, Dr. Jyoti J. Malhotra
- Advanced Java : Dr. Manisha Bharambe
- Android Programming : Kamil Ajmal Khan

BOOKS AVAILABLE AT

PRAGATI BOOK CENTER - Email: pbcpune@pragationline.com

- 157 Budhwar Peth, Opp. Ratan Talkies, Next To Balaji Mandir, **Pune** 411002 • **Mobile** : 9657703148
- 676/B Budhwar Peth, Opp. Jogeshwari Mandir, **Pune** 411002 Tel : (020) 2448 7459 • **Mobile** : 9657703147 / 9657703149
- 152 Budhwar Peth, Near Jogeshwari Mandir, **Pune** 411002 **Mobile** : 8087881795

PRAGATI BOOK CORNER - Email: niralimumbai@pragationline.com

- Apurva Building, Shop No. 1, Bhavani Shankar Road, Opp. Shardashram Society, Dadar (W), Mumbai 400028. Tel: (022) 2422 3526/6662 5254 • Mobile : 9819935759

niralipune@pragationline.com | www.pragationline.com

Also find us on www.facebook.com/niralibooks @nirali.prakashan

ND6188

ISBN 93545146 9-3

9 789354 514692