## UNIT-3-TESTING

## Software Testing:

It is the process used to help identify the correctness, completeness, security, and quality of developed computer software.

Testing is a process of technical investigation, performed on behalf of stakeholders, that is intended to reveal quality-related information about the product with respect to the context in which it is intended to operate. Testing furnishes a criticism or comparison that compares the state and behavior of the product against a specification.

Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Although crucial to software quality and widely deployed by programmers and testers, software testing still remains an art, due to limited understanding of the principles of software. The difficulty in software testing stems from the complexity of software: we can not completely test a program with moderate complexity. Testing is more than just debugging.

The purpose of testing can be quality assurance, verification and validation, or reliability estimation. Testing can be used as a generic metric as well. Correctness testing and reliability testing are two major areas of testing. Software testing is a trade-off between budget, time and quality.

# Testing Objective

Glen Myers [MYE79] states a number of rules that can serve well as testing objective:

* Testing is the process of executing a program with the intent of finding an error.
* A good test case is one that has a high probability of finding an as yet undiscovered error.
* A successful test is one that uncovers an as –yet-undiscovered error.

Regardless of the limitations, testing is an integral part in software development. It is broadly deployed in every phase in the software development cycle. Typically, more than 50% percent of the development time is spent in testing. Testing is usually performed for the following purposes:

* **To improve quality.**

As computers and software are used in critical applications, the outcome of a bug can be severe. Bugs can cause huge losses. Bugs in critical systems have caused airplane crashes, allowed space shuttle missions to go awry, halted trading on the stock market, and worse. Bugs can kill. Bugs can cause disasters. The so-called year 2000 (Y2K) bug has given birth to a cottage industry of consultants and programming tools dedicated to making sure the modern world doesn't come to a screeching halt on the first day of the next century. In a computerized embedded world, the quality and reliability of software is a matter of life and death.

Quality means the conformance to the specified design requirement. Being correct, the minimum requirement of quality, means performing as required under specified circumstances. Debugging, a narrow view of software testing, is performed heavily to find out design defects by the programmer. The imperfection of human nature makes it almost impossible to make a moderately complex program correct the first time. Finding the problems and get them fixed is the purpose of debugging in programming phase.

* **For Verification & Validation (V&V)**

Just as topic Verification and Validation indicated, another important purpose of testing is verification and validation (V&V). Testing can serve as metrics. It is heavily used as a tool in the V&V process. Testers can make claims based on interpretations of the testing results, which either the product works under certain situations, or it does not work. We can also compare the quality among different products under the same specification, based on results from the same test.

We can not test quality directly, but we can test related factors to make quality visible. Quality has three sets of factors --  functionality, engineering, and adaptability. These three sets of factors can be thought of as dimensions in the software quality space. Each dimension may be broken down into its component factors and considerations at successively lower levels of detail. Table 1 illustrates some of the most frequently cited quality considerations.   
 Tests with the purpose of validating the product works are named clean tests, or positive tests. The drawbacks are that it can only validate that the software works for the specified test cases. A finite number of tests cannot validate that the software works for all situations. On the contrary, only one failed test is sufficient enough to show that the software does not work. Dirty tests, or negative tests, refer to the tests aiming at breaking the software, or showing that it does not work. A piece of software must have sufficient exception handling capabilities to survive a significant level of dirty tests.

A testable design is a design that can be easily validated, falsified and maintained. Because testing is a rigorous effort and requires significant time and cost, design for testability is also an important design rule for software development.

* **For reliability estimation**

Software reliability has important relations with many aspects of software, including the structure, and the amount of testing it has been subjected to. Based on an operational profile (an estimate of the relative frequency of use of various inputs to the program testing can serve as a statistical sampling method to gain failure data for reliability estimation.

Software testing is not mature. It still remains an art, because we still cannot make it a science. We are still using the same testing techniques invented 20-30 years ago, some of which are crafted methods or heuristics rather than good engineering methods. Software testing can be costly, but not testing software is even more expensive, especially in places that human lives are at stake. Solving the software-testing problem is no easier than solving the Turing halting problem. We can never be sure that a piece of software is correct. We can never be sure that the specifications are correct. No verification system can verify every correct program. We can never be certain that a verification system is correct either.

## Testing Principles:

Davis [DAV95] suggests a set of testing principles that have been adapted for use:

* All Tests should be traceable to customer requirements:

The objective of software testing is to uncover errors. It follows that the most server defects (from the customer‘s point of view) are those that cause the program to fail to meet its requirements.

* Test should be planned long before testing begins:

Test planning can begin as soon as the design model has been solidified .therefore all tests can be planned and designed before any code has been generated.

* The pareto principle applies to software testing:

The Pareto principle implies that 80% of all errors uncovered during testing will likely be traceable to 20% of all program components. The problem is course, is to isolate these suspect components and to thoroughly test them.

* Testing should begin “in the small” and progress toward testing “in the large ”:

The first test planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

* Exhaustive testing is not possible:

The number of path permutation for even a moderately sized program is exceptionally large .For this reasons, it is impossible to execute every combination of paths during testing . It is possible, however, to adequately cover program logic and to ensure that all conditions in the component –level design have been exercised.

* To be most effective, testing should be conducted by an independent third party: By most effective, we mean testing that has the highest probability of finding errors (the primary objective of testing).The software engineering who has created the system is not the best persons to conduct all tests for the software.
* Testing time and resources are limited:

Avoid redundant tests.

* Keep software static during test:

The program must not be modified during the implementation of the set of designed test cases.

# Software Testing Principles

Software testing is an extremely creative and intellectually challenging task. When testing follows the principles given below, the creative element of test design and execution rivals any of the preceding software development steps.

1. **Testing must be done by an independent party.**   
   Testing should not be performed by the person or team that developed the software since they tend to defend the correctness of the program.
2. **Assign best personnel to the task.**   
   Because testing requires high creativity and responsibility only the best personnel must be assigned to design, implement, and analyze test cases, test data and test results.
3. **Testing should not be planned under the tacit assumption that no errors will be found.**
4. **Test for invalid and unexpected input conditions as well as valid conditions.**  
   The program should generate correct messages when an invalid test is encountered and should generate correct results when the test is valid.
5. **The probability of the existence of more errors in a module or group of modules is directly proportional to the number of errors already found.**
6. **Testing is the process of executing software with the intent of finding errors.**
7. **Keep software static during test.**  
   The program must not be modified during the implementation of the set of designed test cases.
8. **Document test cases and test results.**
9. **Provide expected test results if possible.**   
   A necessary part of test documentation is the specification of expected results, even if providing such results is impractical.

# Testability:

* Operability
* Observability
* Controllability
* Decomposability
* Simplicity
* Stability
* Understability

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

* The system has few bugs (bugs add analysis and reporting overhead to the process).
* No bugs block the execution of tests.
* The product evolves in functional stages (allow simultaneous development and testing).

Observability: “what you see is what you test.”

* Distinct output is generated for each input.
* System states and variables are visible or queriable during execution.
* Past system states and variables are visible or queriable (e.g. transaction logs).
* All factors affecting the output are visible.
* Incorrect output is easily identified.
* Internal errors are automatically detected through self-testing mechanisms.
* Source code is accessible.

Controllability: “The better we can control the software, the more the testing can be automated and optimized.”

* All possible outputs can be generated through some combination of input.
* All code is executable through some combination of input.
* Software and hardware states and variables can be controlled directly by the test engineer.
* Input and output formats are consistent and structured.

Decomposability: “By controlling the scope of testing, we can more quickly isolated problems and perform smarter retesting.”

* The software system is built from independent modules.
* Software modules can be tested independently.

Simplicity: “the less there is to test, the more quickly we can test it.”

* Functionality simplicity (e.g. the feature set is the minimum necessary to meet requirements.).
* Structural simplicity (e.g. architecture is modularized to limit the propagation of faults).
* Code simplicity (e.g. coding standard is adopted for ease of inspection and maintenance ).

Stability: “the fewer the changes, the fewer the disruptions to testing.”

* Changes to the software are infrequent.
* Changes to the software are controlled.
* Changes to the software do not invalidate existing tests.
* The software recovers well from failure.

Understandability: “the more information we have, the smarter we will test.”

* The design is well understood.
* Dependencies between internal, external, and shared components are well understood.
* Changes to the design are communicated.
* Technical documentation is instantly accessible.
* Technical documentation is well organized.
* Technical documentation is specific and detailed.
* Technical documentation is accurate.

## What are the attributes of a “good” test? :

1. **A good test a high probability of finding an error:**

To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail. The classes of failure are probed.

**For example:** one class of potentially failure in GUI (graphical user interface) is a failure to recognize proper mouse position.

1. **A good test is not redundant:**

Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test.

**For example:** A module of the safe home software is designed to recognize a user password to activate and deactivate the system.

The invalid password 1234 should not be accepted by a system programmed to recognize 8080 as the valid password. If it is accepted, an error is present. Another test input, 1235, would have the same purpose as 1234 and is therefore redundant.

1. **A good test should be “best of breed”:**

In a group of test that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases the test that has the highest likelihood of uncovering a whole class of errors should be used.

1. **A good test should be neither too simpler nor too complex:**

It is sometime possible to combine a series of tests into one test case; the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

## White box testing:

White box sometimes called glass box testing. Contrary to black-box testing, software is viewed as a white-box, or glass-box in white-box testing, as the structure and flow of the software under test are visible to the tester. Testing plans are made according to the details of the software implementation, such as programming language, logic, and styles. Test cases are derived from the program structure. White-box testing is also called glass-box testing, logic-driven testing or design-based testing .

There are many techniques available in white-box testing, because the problem of intractability is eased by specific knowledge and attention on the structure of the software under test. The intention of exhausting some aspect of the software is still strong in white-box testing, and some degree of exhaustion can be achieved, such as executing each line of code at least once (statement coverage), traverse every branch statements (branch coverage), or cover all the possible combinations of true and false condition predicates (Multiple condition coverage).

Control-flow testing, loop testing, and data-flow testing, all maps the corresponding flow structure of the software into a directed graph. Test cases are carefully selected based on the criterion that all the nodes or paths are covered or traversed at least once. By doing so we may discover unnecessary "dead" code -- code that is of no use, or never get executed at all, which can not be discovered by functional testing.

Using white box testing methods the software engineer can test cases that:

* Guarantee that all independent paths within a module have been exercised at least once.
* Exercise all loops at their boundaries.
* Exercise all logical decision on their true and false sides.
* Exercise internal data structures to ensure their validity.

Reasons why white box carried out:

White box testing is carried out to test whether

* All paths in a process are correctly operational.
* All logical decision is executed with true and false conditions.
* All loops are executed with their limit values tested

Advantages of structural/ white box testing:

* Forces test developer to reason carefully about implementation.

White box testing

White box testing

## Structural Testing:

White Box Test is also known by other names such as glass-box testing, structural testing, clear box testing, open-box testing, and path-oriented testing.

Text data

Texts datives

text output

Component code

Structural testing

Methods/approaches of structural testing

* Condition testing
* Dataflow testing
* Loop testing

Condition testing:-is a test case design method that exercise the logical condition contained in a program module. A simple condition is a Boolean variable or a relational expression possible preceded with one NOT operator (¬) .A relational expression takes the form

“E1 <relational-operator>E2”

Where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following condition following :<,<=,=,≠,>,>=. A compound condition is composed of two or more simple conditions, Boolean operators and paranthesis.Boolean operators include:

OR(І)Í  
AND(&)

NOT (¬)

If a condition is incorrect, then at least one component of the condition is incorrect. therefore, types of errors in a condition include the following:

* Boolean operator error(incorrect /missing/extra Boolean operators)
* Boolean variable error
* Boolean parenthesis error
* Relational operator error
* Arithmetic expression error

**Purpose: To detect not only errors in the conditions of a program but also other errors in the program. If a test set for a program p is effective for detecting errors in the condition contained in p, it is likely that this test set is also effective for detecting other errors in p.**

**A number of condition testing strategies have been proposed:**

* **Branch testing:** is a probably the simplest condition testing strategy for a compound condition C, the true and false branches of c and very simple condition in C need to be executed at least once.
* **Domain testing: I**t requires 3-4 tests to be derived for a relational expression.

**Data Flow Testing**

The data flow testing method selects test paths of program according to the locations of definition and uses of variables in the program. A number of data flow testing strategies have been studied and compared.

To define the data flow testing it is assumed that each statement in a program is assigned a unique statement number. For a statement with S as its statement numbers

Def(S) = {A/Statement S contains a definition of A}

Use(S’) = {A/Statement S contains a use of A}

if statement S is an if or loop statement, Its Def set is empty and its use set is based on the condition of statement S.

A def use chain of variable A is of the form {A,S,S’}where S and S’ are the statement numbers, A is in Def(S) and use (S’),and definition of A in statement S is live at statement S’.

**Loop testing**

Loop testing is a white box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined:

* **Simple loops:** the following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

1. **Skip the loop entirely.**
2. **Only one pass through the loop.**
3. **Two passes through the loop.**
4. **m passes through the loop where m<n.**
5. **n-1,n,n+1 passes through the loop.**

Simple loops

* **Nested Loops: I**f we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests.
  1. Start at the innermost loop. Set all other loops to minimum values.
  2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter.
  3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to “typical” values.
* Continue until all loops have been tested.
* **Concatenated loops:** Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other.
* **Unstructured loops: W**henever possible, this class of loops should be re designed to reflect the use of the structured programming constructs**.**

## Black box Testing/functional testing:

Functional testing refers to testing, which involves only observation of the output for certain input values, and there is no attempt to analyze the code, which produces the output. The internal structure of the program is ignored. For this reason, functional testing is sometimes referred to as a black box testing in which the content of a black box is not known and function of black box is understood completely in terms of its inputs and outputs.

Black-box testing is not an alternative to white-box techniques; rather it is complementary approach that is likely to uncover a different class of errors than white-box methods.

Black box testing identifies the following the following kind of errors:

* In corrector missing functions.
* Interface missing or erroneous.
* Error in data model.
* Error in access to external data source.

When these errors are controlled then:

* Function(s) are valid.
* A class of inputs is validated.
* Validated is sensitive to certain input values.
* The software is valid and dependable for certain volume of data or transactions.
* Rare specific combinations are taken care of.

black box testing tries to answer the following questions:

* How is functional validity tested?
* How are system behavior & performance tested?
* How are the boundaries of a data class isolated?
* How the test cases will make good?
* How the specific combination of data will affect the system operation?
* What data rates & data volume can the system tolerate?
* Is the system particularly sensitive to certain input values?

By applying black-box techniques, we derive set test cases that satisfy the following criteria:

* Test cases that reduce, by a count that in greater than one.
* Test cases that tell us something about the presence or absence of classes of errors.

**Categories of black box testing:**

There are two categories of testing:

* Positive functional testing
* Negative functional testing

**Positive functional testing:**

This testing entails exercising the application’s function with valid input and verifying that the outputs are correct.

**Example:** continuing with the word processing example, a positive test for the printing function might be to print a document containing both text and graphics to a printer that is online, filled with paper and or which the correct drivers are installed.

**Negative functional testing:**

This testing involves exercising application functionality using a combination of invalid inputs, unexpected operating condition and other “out-of-bounds scenarios”.

**Example:**

1. Containing the word processing example, a negative test for the printing function might be to disconnect the printer from the computer while a document is printing.
2. What might happen instead is the word processing software simply hangs up or crashes because the “abnormal” loss of communications with the printer is not handled properly.

**Advantages of black box testing:**

* The test is unbiased because the designer and the tester are independent of each other.
* The tester does not need knowledge of any specific programming languages.
* The test is done from the point of view of the user, not the designer.
* Test cases can be designed as soon as the specifications are complete.

**Functional/black box testing approaches:**

* **Graph based testing**
* **Equivalence partitioning**
* **Boundary value analysis(BVA)**
* **Orthogonal array testing**

**Graph based testing:**

* Software testing beings by creating a graph of important objects and their relationship and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.
* To accomplish these steps, the software engineer beings by creating a graph, i.e., a collection of nodes that represent objects, links that represent the relationship between objects, node weights that describe the properties of a node, and link weights that describe some characteristics of a link.
* Nodes are represented as circle connected by links that take a number of different forms. A directed link indicates that a relationship moves in only one direction.

Directed links

Object 1 object object2

Undirected links parallel links

Object 3

Graph notation

Menu select generates

New file menu document window

Select

Is represented as contains

Document text

Simple example

**Equivalence class testing:**

* If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
* If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
* If an input condition specifies a member of a set, one valid and one invalid equivalence classes are defined.
* If an input condition is Boolean, one invalid class is defined.

By applying these guidelines for the derivation of an equivalence classes, test cases for each input domain data object can be developed and executed.

Valid input system

inin

outputs

invalid inputs

equivalence partitioning

**Boundary value analysis:**

* A greater number of errors occur at the boundaries of the input domain rather than in the center.
* It is for this reason that boundary value analysis has been developed as a testing technique. BVA leads to a selection of test cases that exercised boundary values.
* Boundary value analysis is a test case design technique that complements equivalence partitioning.
* Rather than selecting any element of the equivalence class, BVA leads to the selection of test cases at the edges of the class.
* Rather than focusing solely on input conditions, BVA derivers test cases from the output domain as well.

Guidelines for BVA are:

* If an input condition specifies a range bounded by value a and b, test cases should be designed with values a and b as well as just above and just below a and b.
* If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
* If internal program data structures have prescribed boundaries, be certain to design a test case to exercise the data structure at its boundary.
* Most of the software engineer intuitively performs BVA to some degree. By applying these guidelines boundary testing will be more complete, thereby having a higher likelihood for error detection.

**Orthogonal array testing:**

* There are many applications in which the input domain is relatively limited.
* Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodated exhaustive testing.
* The orthogonal array testing method is particularly useful in finding errors, associated with region faults that are an error category associated with faulty logic within a software component.
* To illustrate the difference between orthogonal array testing and more conventional “one input time at a time” approaches, consider a system that has three input items x,y,and z.
* Each of these input item has three discrete values associated with it . there are cube(3)=27 possible test cases.

## Functional Testing

## Stress Testing

## Load Testing

## Functional Testing

## In this type of testing, the software is tested for the functional requirements. The tests are written in order to check if the application behaves as expected. Although functional testing is often done toward the end of the development cycle, it can—and should, —be started much earlier. Individual components and processes can be tested early on, even before it's possible to do functional testing on the entire system. Functional testing covers how well the system executes the functions it is supposed to execute—including user commands, data manipulation, searches and business processes, user screens, and integrations. Functional testing covers the obvious surface type of functions, as well as the back-end operations (such as security and how upgrades affect the system).

## Stress Testing

The application is tested against heavy load such as complex numerical values, large number of inputs, large number of queries etc. which checks for the stress/load the applications can withstand. Stress testing deals with the quality of the application in the environment.   The idea is to create an environment more demanding of the application than the application would experience under normal work loads. This is the hardest and most complex category of testing to accomplish and it requires a joint effort from all teams. A test environment is established with many testing stations. At each station, a script is exercising the system. These scripts are usually based on the regression suite. More and more stations are added, all simultaneous hammering on the system, until the system breaks. The system is repaired and the stress test is repeated until a level of stress is reached that is higher than expected to be present at a customer site. Race conditions and memory leaks are often found under stress testing. A race condition is a conflict between at least two tests. Each test works correctly when done in isolation. When the two tests are run in parallel, one or both of the tests fail. This is usually due to an incorrectly managed lock. A memory leak happens when a test leaves allocated memory behind and does not correctly return the memory to the memory allocation scheme. The test seems to run correctly, but after being exercised several times, available memory is reduced until the system fails.

**Load Testing**

## The application is tested against heavy loads or inputs such as testing of web sites in order to find out at what point the web-site/application fails or at what point its performance degrades. Load testing operates at a predefined load level, usually the highest load that the system can accept while still functioning properly. Note that load testing does not aim to break the system by overwhelming it, but instead tries to keep the system constantly humming like a well-oiled machine.In the context of load testing, extreme importance should be given of having large datasets available for testing. Bugs simply do not surface unless you deal with very large entities such thousands of users in repositories such as LDAP/NIS/Active Directory; thousands of mail server mailboxes, multi-gigabyte tables in databases, deep file/directory hierarchies on file systems, etc. Testers obviously need automated tools to generate these large data sets, but fortunately any good scripting language worth its salt will do the job.

## Test plan:

A test plan documents the strategy that will be used to verify and ensure that a product or system meets its design specifications and other requirements. A test plan is usually prepared by or with significant input from Test Engineers.

Depending on the product and the responsibility of the organization to which the test plan applies, a test plan may include one or more of the following:

* Design Verification or Compliance test - to be performed during the development or approval stages of the product, typically on a small sample of units.
* Manufacturing or Production test - to be performed during preparation or assembly of the product in an ongoing manner for purposes of performance verification and quality control.
* Acceptance or Commissioning test - to be performed at the time of delivery or installation of the product.
* Service and Repair test - to be performed as required over the service life of the product.
* Regression test - to be performed on an existing operational product, to verify that existing functionality didn't get broken when other aspects of the environment are changed (e.g., upgrading the platform on which an existing application runs).

A test plan should contain the following:

* Test unit specification
* Features to be tested
* Approach for testing
* Test deliverables
* Scheduled
* Personnel allocation

**Test unit specification:**

A test unit is a set of one or more modules, together with associated data, that are from a single computer program and that are the object of testing.

**Features to be tested:**

Features to be tested include all software features and combination of features that should be tested. A software features is a software characteristic specified or implied by the requirements or design documents.

**The approaches for testing:**

The techniques that will be used to judge the testing effort should also be specified. This is some time called the testing criterion or criterion for evaluating the set of test cases used in testing.

**Testing deliverable:**

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

**Schedule**:

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

## Test case specification:

Test case specification has to be done separately for each unit. Based on the approach specified in the test plan, first the features to be tested for this unit must be determined. The overall approach stated in the plan is refined into specific test techniques that should be that should be followed and into the criteria to be used for evaluation. Based on these, the test cases are specified for testing the units. Test case specification gives, for each unit to be tested, all the test cases inputs to be used in the test cases, conditions being tested by the test case, and outputs expected for those test cases.

Test plan focuses on how the testing for the project will proceed, which unit will be tested. Test case specification has to be done separately for each unit. Test case specification document is reviewed using a formal review process, to make sure that the test cases are consistent with the policy specified in the plan satisfy the chosen criterion and in general cover the various aspects of the unit to be tested. Specifications can be used as scripts during regressions testing.

## Software testing strategic

A software strategy should be flexible enough to promote a customized testing approach. At the same time, it must be rigid enough to promote reasonable planning and management tracking as the project progresses.

**Strategic characteristic:**

* Testing begins at the component level and works “outward” toward the integration of the entire computer-based system.
* Different testing techniques are appropriate at different points in time.
* Testing is conducted by the developer of the software and (for large projects) an independent test group.
* Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.
* A strategy for software testing must accommodate low-level tests that are necessary to verify that a small code segment has been correctly implemented as well as high –level tests that validate major system functions against customer requirements.

**It has many approaches**:

1. Verification and validation
2. Formal technical review
3. Walk through
4. Code inspection
5. Compilation with design and coding standards

**Verification and validation:**

Verification refers to the set of activities that ensure that software correctly implements a specific function.

Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.

“Verification: “are we building the product right?””

“Validation: “are we building the right product?””

Just as topic Verification and Validation indicated, another important purpose of testing is verification and validation (V&V). Testing can serve as metrics. It is heavily used as a tool in the V&V process. Testers can make claims based on interpretations of the testing results, which either the product works under certain situations, or it does not work. We can also compare the quality among different products under the same specification, based on results from the same test.

We can not test quality directly, but we can test related factors to make quality visible. Quality has three sets of factors -- functionality, engineering, and adaptability. These three sets of factors can be thought of as dimensions in the software quality space. Each dimension may be broken down into its component factors and considerations at successively lower levels of detail.

**Formal technical review:**

It is a software quality assurance activity performed by software engineering practitioners to improve software product quality. The product is scrutinized for completeness, corrections, consistency, technical feasibility, efficiency, and adherence to establish standards and guidelines by the client organization.

**Objective of formal technical review:**

* To uncovererrors in logic and implementation.
* To ensure that the software has been represented accruing to predefined standards.
* To ensure that software under review, meets the requirements.
* To make the project more manageable

**Code walk through:**

A code walk through is an informal analysis of code as a cooperative, organized activity by several participants. The analysis is based mainly on the game of “playing the computer”. That is, participation selects some test cases and simulation execution of the code by hand. This is the reasons for the name “walk-through the code” or through any design notation.

* Everyone’s work should be reviewed on a scheduled basis
* The number of people involved in the review should be small (three or five)
* The participants should receive written documentation from the designer a few days before the meeting.
* The meeting should last a predefined amount of time (a few hours).

**Advantages of code walk through**:

1. It improves project team communication and project morale by involving team members in the walkthrough process.
2. The earlier bugs are found in the life cycle of a product, the cheaper and easier they are to fix.

**Disadvantage**:

The disadvantage of code walk through is that they take time, not only of the person actively working on the project, but also for other people who are usually under deadline pressure themselves.

**Code inspection:**

A code inspection originally introduced by Fagan (1976) at IBM, is similar to a walk through but is more formal. Fagan’s experiment, three separate inspections were performed:

1. One following design, but prior to implementation.
2. One following implementation, but prior to unit testing.
3. One following unit testing

The inspection following unit testing was not considered to be cost effective in discovering errors, therefore it is not recommended.

An inspection team consists of four persons, who play the roles of moderator, reader, recorder and author.

Moderator: - the moderator must be technically very competent. He is the person who leads the inspection process.

Reader: - the reader takes the team through the code by paraphrasing its operation.

RECODER:-A recorder notes each error on a standard form. This frees the other team members to focus on thinking deeply about the code.

Authors: - the author’s role is to understand the errors found and to illuminate unclear areas. As code inspections are never confrontational, the author should never be in a position of defending the code.

The following is a list of some classical programming errors, which can be checked during code inspection:

* Use of un-initialized variables.
* Jumps into loops
* Non –terminating loops
* Incompatible assignments
* Array indices out of bounds
* Improper storage allocation and deal location
* Improper modification of loop variables
* Comparison of equality of floating-point values, etc.

**Advantages of code inspection:**

1. Lists all potential design flows that can make software code less maintainable and costly to develop.
2. Indicates all defects otherwise difficult to testing and usage.
3. A detailed error feedback is provided to individual programmers. So that it makes easier for them to make changes in the code.

## Strategic issues:

* Specify product requirements in a quantifiable manner long before testing commences.
* State testing objectives explicitly.
* Understand the users of the software and develop a profile for each user category.
* Develop a testing plan that emphasizes “rapid cycle testing.”
* Build “robust” software that is designed to test itself.
* Use effective formal technical reviews as a filter prior to testing.
* Conduct formal technical reviews to assess the test strategy and test cases themselves.

## Level of testing:

1. **Unit testing**
2. **Integration testing**
3. **System testing**
4. **User acceptance testing**
5. **Regression Testing**

**Unit testing:-**

The developer carries out unit testing in order to check if the particular module or unit of code is working fine. The 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 testing deals with testing a unit as a whole. This would test the interaction of many functions but confine the test within one unit. The exact scope of a unit is left to interpretation. Supporting test code, sometimes called scaffolding, may be necessary to support an individual test. This type of testing is driven by the architecture and implementation teams. This focus is also called black-box testing because only the details of the interface are visible to the test. Limits that are global to a unit are tested here. In the construction industry, scaffolding is a temporary, easy to assemble and disassemble, frame placed around a building to facilitate the construction of the building. The construction workers first build the scaffolding and then the building. Later the scaffolding is removed, exposing the completed building. Similarly, in software testing, one particular test may need some supporting software. This software establishes an environment around the test. Only when this environment is established can a correct evaluation of the test take place. The scaffolding software may establish state and values for data structures as well as providing dummy external functions for the test. Different scaffolding software may be needed from one test to another test. Scaffolding software rarely is considered part of the system. Sometimes the scaffolding software becomes larger than the system software being tested. Usually the scaffolding software is not of the same quality as the system software and frequently is quite fragile. A small change in the test may lead to much larger changes in the scaffolding. Internal and unit testing can be automated with the help of coverage tools.

A coverage tool analyzes the source code and generates a test that will execute every alternative thread of execution. It is still up to the programmer to combine this test into meaningful cases to validate the result of each thread of execution. Typically, the coverage tool is used in a slightly different way. First the coverage tool is used to augment the source by placing informational prints after each line of code. Then the testing suite is executed generating an audit trail. This audit trail is analyzed and reports the percent of the total system code executed during the test suite. If the coverage is high and the **ntested** source lines are of low impact to the system's overall quality, then no more additional tests are required.

**Integration test:**

Integration testing is a logical extension of unit testing. In its simplest form, two units that have already been tested are combined into a component and the interface between them is tested. A component, in this sense, refers to an integrated aggregate of more than one unit. In a realistic scenario, many units are combined into components, which are in turn aggregated into even larger parts of the program. The idea is to test combinations of pieces and eventually expand the process to test your modules with those of other groups. Eventually all the modules making up a process are tested together. Beyond that, if the program is composed of more than one process, they should be tested in pairs rather than all at once.

Integration testing identifies problems that occur when units are combined. By using a test plan that requires you to test each unit and ensure the viability of each before combining units, you know that any errors discovered when combining units are likely related to the interface between units. This method reduces the number of possibilities to a far simpler level of analysis.

You can do integration testing in a variety of ways but the following are three common strategies:

* The top-down approach to integration testing requires the highest-level modules be test and integrated first. This allows high-level logic and data flow to be tested early in the process and it tends to minimize the need for drivers. However, the need for stubs complicates test management and low-level utilities are tested relatively late in the development cycle. Another disadvantage of top-down integration testing is its poor support for early release of limited functionality.
* The bottom-up approach requires the lowest-level units be tested and integrated first. These units are frequently referred to as utility modules. By using this approach, utility modules are tested early in the development process and the need for stubs is minimized. The downside, however, is that the need for drivers complicates test management and high-level logic and data flow are tested late. Like the top-down approach, the bottom-up approach also provides poor support for early release of limited functionality.
* The third approach, sometimes referred to as the umbrella approach, requires testing along functional data and control-flow paths. First, the inputs for functions are integrated in the bottom-up pattern discussed above. The outputs for each function are then integrated in the top-down manner. The primary advantage of this approach is the degree of support for early release of limited functionality. It also helps minimize the need for stubs and drivers. The potential weaknesses of this approach are significant, however, in that it can be less systematic than the other two approaches, leading to the need for more regression testing.

**Regression testing:**

Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons)do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools. Capture /playback tools and comparison.

The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

* A representative sample of tests that will exercise all software functions.
* Additional tests that focus on software functions that are likely to be affected by the change.
* Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large.

So regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

**Smoke testing:**

Smoke testing is an integration testing approach that is commonly used when “shrink-wrapped” software product are being developed .this testing approach encompasses the following activities:

* Software components that have been translated into code are integrated into a “build”. A build includes all the data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
* A series of tests is designed to exposes errors that will keep the build from properly perform the function.
* The build is integrated with other builds and the entire product smoke tested daily.

**Advantages of smoke testing:**

* **Integration risk is minimized:** because smoke tests are conducted daily, incompatibility and other shows stoppers errors are uncovered early, thereby reducing the likely hood of serious schedule impact when errors are uncovered.
* **The quality of the end-product is improved:** because the approach is construction oriented, smoke testing is likely to uncover both functional errors and architectural and component design defects.
* **Error diagnosis and correctness are simplified:** like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with “new software increments”, -i.e. the software that has just been added to the builds is a probably cause on a newly discovered error.
* **Progress is easier to assess:** with each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

**Integration test documentation**

An overall plan for integration of the software and the description of specific test are documented in a specification. This document contains a test plan, and a test procedure, is a work product of the software process, and becomes the part of the software configuration.

For example, integration testing for a CAD system might be divided into the following test phases:

* User interaction
* Data manipulation
* Display processing and generation
* Database management

These criteria and corresponding tests are applied for all test phases:

* **Interface integrity :** internal and external interfaces are tested as each module is incorporated into the structure.
* **Functional validity :** test designed to uncover functional errors are conducted.
* **Information content:** tests designed to uncover errors associated with local or global data structures are conducted.
* **Performance:** test designed to verify performance bounds established during software designed are conducted.

# Validation testing:

Validation can be defined in many ways, but a simple definition is that validation succeed when software function in a manner that can be reasonably expected by the customer. Information contained in that section forms the basis for a validation testing approach:

A) **Validation test criteria:**

Software validation is achieved through a series of black box tests that demonstrates conformity with requirements. A test plan outlines the classes of tests to be conducted and a test procedure defines specific test cases that will be used to demonstrate conformity with requirements.

After each validation test case has been conducted, one of two possible conditions exists:-

1. The function or performance characteristics conform to specification and are accepted.
2. A deviation from specification is uncovered and a deficiency list is created. Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

B) **Configuration Review:**

An important element of the validation process is a configuration review. The intent of the review is to ensure that all the element of the software configuration have been properly developed , are cataloged, and have necessary detail to bolster the support phase of the software life cycle. The configuration sometime called an audit.

C) **Alpha and Beta testing:**

**The alpha test** is conducted at the developer’s site by a customer. The software is used in a natural setting with the developer “looking over the shoulder” of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

**The Beta test** is conducted at one or more customer sites by the end –user of the software. Unlike alpha testing, the developer is not present .Therefore; the beta test is a “live” application of the software in an environment that cannot be controlled by the developer. The customer records all problem (real or imagined) that are encountered during beta and testing and report these to the developer at regular intervals. As a result of problems reported during beta tests, software engineering makes modification and then prepare for released of the software product to the entire customer base.

# System testing

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer –based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions. The various type of system tests are:-

* Recovery testing
* Security testing
* Stress testing
* Performance testing

**Recovery Testing:**

Recovery testing is basically done in order to check how fast and better the application can recover against any type of crash or hardware failure etc. Type or extent of recovery is specified in the requirement specifications. It is basically testing how well a system recovers from crashes, hardware failures, or other catastrophic problems

**Security testing:**

Software quality, reliability and security are tightly coupled. Flaws in software can be exploited by intruders to open security holes. With the development of the Internet, software security problems are becoming even more severe.

Many critical software applications and services have integrated security measures against malicious attacks. The purpose of security testing of these systems include identifying and removing software flaws that may potentially lead to security violations, and validating the effectiveness of security measures. Simulated security attacks can be performed to find vulnerabilities.

**Stress testing:**

Stress testing deals with the quality of the application in the environment.   The idea is to create an environment more demanding of the application than the application would experience under normal workloads.

### Example:

1. Special tests may be designed that generate ten interrupts per second, when one or two is the average rate.
2. Input data rates may be increased by an order of magnitude to determine how input functions will respond.
3. Tests cases that require maximum memory or other resources are executed.
4. Tests cases that may causes thrashing in a virtual operating system are designed.
5. Tests cases that may causes excessive hunting for disk- resident data are created.

This is the hardest and most complex category of testing to accomplish and it requires a joint effort from all teams. A test environment is established with many testing stations. At each station, a script is exercising the system. These scripts are usually based on the regression suite. More and more stations are added, all simultaneous hammering on the system, until the system breaks. The system is repaired and the stress test is repeated until a level of stress is reached that is higher than expected to be present at a customer site.

Race conditions and memory leaks are often found under stress testing. A race condition is a conflict between at least two tests. Each test works correctly when done in isolation. When the two tests are run in parallel, one or both of the tests fail. This is usually due to an incorrectly managed lock. A memory leak happens when a test leaves allocated memory behind and does not correctly return the memory to the memory allocation scheme. The test seems to run correctly, but after being exercised several times, available memory is reduced until the system fails.

**Performance testing:**

Not all software systems have specifications on performance explicitly. But every system will have implicit performance requirements. The software should not take infinite time or infinite resource to execute. "Performance bugs" sometimes are used to refer to those design problems in software that cause the system performance to degrade.

Performance has always been a great concern and a driving force of computer evolution. Performance evaluation of a software system usually includes: resource usage, throughput, stimulus-response time and queue lengths detailing the average or maximum number of tasks waiting to be serviced by selected resources. Typical resources that need to be considered include network bandwidth requirements, CPU cycles, disk space, disk access operations, and memory usage .The goal of performance testing can be performance bottleneck identification, performance comparison and evaluation, etc. The typical method of doing performance testing is using a benchmark -- a program, workload or trace designed to be representative of the typical system usage.

## Debugging:

Debugging occurs as a consequence of successful testing .i.e., when a test case uncovers an error, debugging is the process that results in the removal of the error.

The various **categories/approaches** of debugging:

1. Brute force categories
2. Backtracking
3. Cause elimination
4. Program slicing
5. Fault-tree analysis

**Brute forced categories:**

The brute forced categories for debugging is probably the most common and least efficient method for isolating the cause of a software error. We apply brute forced debugging method when all else fails using a “let the computer find the error” philosophy, memory dumps are taken, run time traces are invoked, and the program is loaded with WRITE statements.

**Backtracking:**

**It is** fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code traced backward (manually) until the site of the causes is found.

**Cause elimination:**

**Cause elimination** is manifested by induction or deduction and introduces the concepts of binary partitioning. Data related to the error occurrence are orgabized to isolate potential causes.

**A list** of all possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular causes hypothesis shows promises; the dta are refined in an attempt to isolate the bug.

**Program slicing:**

This technique is similar to backtracking. However, the search space is reduced by defining slices. A slice of program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.

**Fault –tree Analysis:**

**It a method originally** developed for the U.S. Minute-man missile program, reasons about the design helping us to decompose it and look for situation that might lead to failure. s

## The debugging process:

**Debugging** is not testing but always occurs as consequences of testing. The debugging process will always have one or more outcomes:

1. The causes will be found and corrected.
2. The causes will not be found.

In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

Some few characteristics of bugs provide some clues:

* The symptom may disappear (temporally) when another error is corrected.
* The symptom may actually be caused by non errors (round-off, inaccuracies.).
* The symptom may caused by human error that is not easily traced.
* The symptom may be a result of timing problems, rather than processing problems.
* The symptom may be intermittent. This is practically common in embedded system that couple hardware and software inextricably.
* The symptom may be due to causes that are distributed across a number of tasks running on different processors.
* It may be difficult to accurately reproduced input conditions.

## Program debugging:

The program testing and debugging is same thing. Though closely related, but they are distinct processes. Debugging is depends on the output of testing which tells the programmer about presences or absence of errors.

There are various debugging stages:

Reset the modified program

Repair the error found

Design the error repair

Locate the error

**Debugging Stages**

The debugger must have knowledge of the common errors, which occur very often in program. After has error been discovered, then correct the error. If the error is some coding error, then that error can be corrected easily. But, if the error is some design mistake then it may require efforts and time. Program listing and the hard copy of the output can be an aid to debugger.

## Debugging guidelines:

Some general guidelines for effective debugging are:

* Many a times, debugging require a thorough understanding of the program design.
* Debugging may sometimes even require full redesign of the system.
* One must be beware of the possibility that any error correction may introduce new errors. So, after every round of error –fixing, regression testing must be carried out.