K.D. K. College of Engineering

Department of Information Technology

B.E. (Information Technology) Fifth Semester (C.B.S.)

Software Engineering

UNIT: 05

UNIT Testing

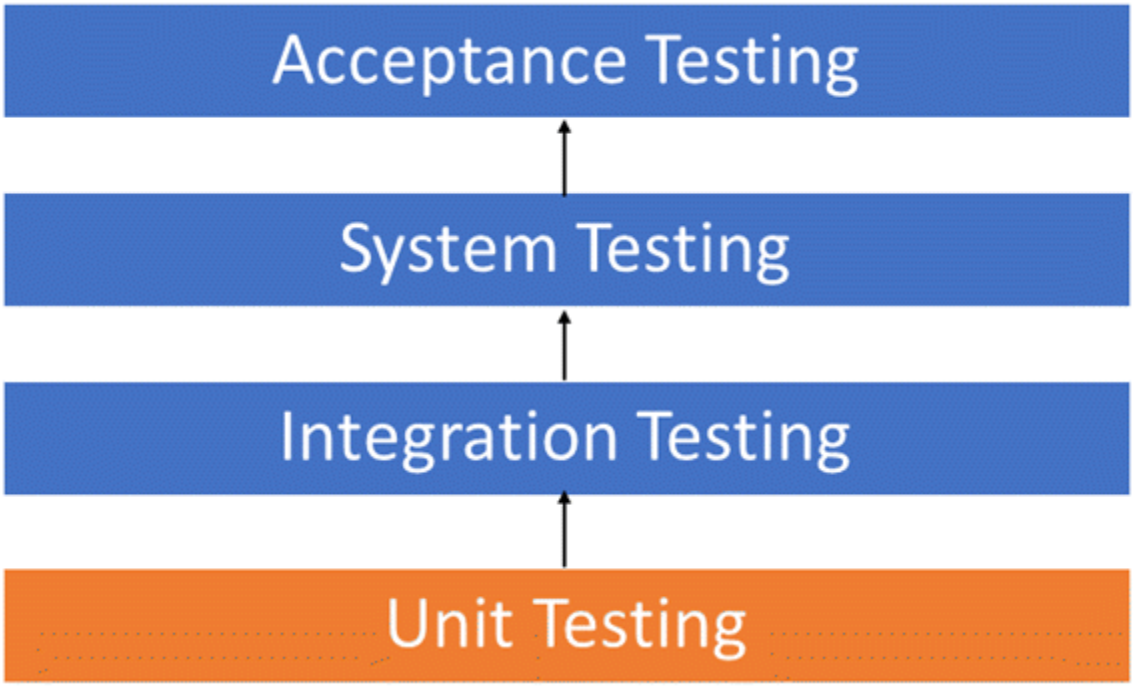
UNIT Testing is defined as a type of software testing where individual units/ components of a software are tested.

Unit Testing of software applications is done during the development (coding) of an application. The objective of Unit Testing is to isolate a section of code and verify its correctness. In procedural programming, a unit may be an individual function or procedure. Unit Testing is usually performed by the developer.

In SDLC, STLC, V Model, Unit testing is first level of testing done before integration testing. Unit testing is a WhiteBox testing technique that is usually performed by the developer. Though, in a practical world due to time crunch or reluctance of developers to tests, QA engineers also do unit testing.

Why Unit Testing?

Sometimes software developers attempt to save time by doing minimal unit testing. This is a myth because skipping on unit testing leads to higher Defect fixing costs during System Testing, Integration Testing and even Beta Testing after the application is completed. Proper unit testing done during the development stage saves both time and money in the end. Here, are key reasons to perform unit testing.



1. Unit Tests fix bug early in development cycle and save costs.
2. It helps understand the developers the code base and enable them to make changes quickly
3. Good unit tests serve as project documentation
4. Unit tests help with code re-use. Migrate both your code andyour tests to your new project. Tweak the code till the tests run again.

How to do Unit Testing

Unit Testing is of two types

- Manual
- Automated

Unit testing is commonly automated but may still be performed manually. Software Engineering does not favor one over the other but automation is preferred. A manual approach to unit testing may employ a step-by-step instructional document.

Under the automated approach-

- A developer writes a section of code in the application just to test the function. They would later comment out and finally remove the test code when the application is deployed.
- A developer could also isolate the function to test it more rigorously. This is a more thorough unit testing practice that involves copy and paste of code to its own testing environment than its natural environment. Isolating the code helps in revealing

unnecessary dependencies between the code being tested and other units or data spaces in the product. These dependencies can then be eliminated.

- A coder generally uses a UnitTest Framework to develop automated test cases. Using an automation framework, the developer codes criteria into the test to verify the correctness of the code. During execution of the test cases, the framework logs failing test cases. Many frameworks will also automatically flag and report, in summary, these failed test cases. Depending on the severity of a failure, the framework may halt subsequent testing.
- The workflow of Unit Testing is 1) Create Test Cases 2) Review/Rework 3) Baseline 4) Execute Test Cases.

Unit Testing Techniques

Code coverage techniques used in united testing are listed below:

- Statement Coverage
- Decision Coverage
- Branch Coverage
- Condition Coverage
- Finite State Machine Coverage

Unit Testing Example: Mock Objects

Unit testing relies on mock objects being created to test sections of code that are not yet part of a complete application. Mock objects fill in for the missing parts of the program.

For example, you might have a function that needs variables or objects that are not created yet. In unit testing, those will be accounted for in the form of mock objects created solely for the purpose of the unit testing done on that section of code.

Unit Testing Advantage

- 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 API.
- Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly (i.e. Regression testing). The procedure is to write test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified and fixed.
- Due to the modular nature of the unit testing, we can test parts of the project without waiting for others to be completed.

Unit Testing Disadvantages

- Unit testing can't be expected to catch every error in a program. It is not possible to evaluate all execution paths even in the most trivial programs

- Unit testing by its very nature focuses on a unit of code. Hence it can't catch integration errors or broad system level errors.

It's recommended unit testing be used in conjunction with other testing activities.

Integration Testing

Integration testing is the process of testing the interface between two software units or module. It's focus on determining the correctness of the interface. The purpose of the integration testing is to expose faults in the interaction between integrated units. Once all the modules have been unit tested, integration testing is performed.

Integration test approaches –
There are four types of integration testing approaches. Those approaches are the following:
1. Big-Bang Integration Testing –
It is the simplest integration testing approach, where all the modules are combining and verifying the functionality after the completion of individual module testing. In simple words, all the modules of the system are simply put together and tested. This approach is practicable only for very small systems. If once an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated. So, debugging errors reported during big bang integration testing are very expensive to fix.
Advantages:

- It is convenient for small systems.

Disadvantages:

- There will be quite a lot of delay because you would have to wait for all the modules to be integrated.
- High risk critical modules are not isolated and tested on priority since all modules are tested at once.

2. Bottom-Up Integration Testing –
In bottom-up testing, each module at lower levels is tested with higher modules until all modules are tested. The primary purpose of this integration testing is, each subsystem is to test the interfaces among various modules making up the subsystem. This integration testing uses test drivers to drive and pass appropriate data to the lower level modules.
Advantages:

- In bottom-up testing, no stubs are required.
- A principle advantage of this integration testing is that several disjoint subsystems can be tested simultaneously.

Disadvantages:

- Driver modules must be produced.
- In this testing, the complexity that occurs when the system is made up of a large number of small subsystem.

3. Top-Down Integration Testing –
Top-down integration testing technique used in order to simulate the behaviour of the lower-

level modules that are not yet integrated.In this integration testing, testing takes place from top to bottom. First high-level modules are tested and then low-level modules and finally integrating the low-level modules to a high level to ensure the system is working as intended.
Advantages:

- Separately debugged module.
- Few or no drivers needed.
- It is more stable and accurate at the aggregate level.

Disadvantages:

- Needs many Stubs.
- Modules at lower level are tested inadequately.

4. Mixed Integration Testing –
A mixed integration testing is also called sandwiched integration testing. A mixed integration testing follows a combination of top down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level module have been coded and unit tested. In bottom-up approach, testing can start only after the bottom level modules are ready. This sandwich or mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. A mixed integration testing is also called sandwiched integration testing.
Advantages:

- Mixed approach is useful for very large projects having several sub projects.
- This Sandwich approach overcomes this shortcoming of the top-down and bottom-up approaches.

Disadvantages:

- For mixed integration testing, require very high cost because one part has Top-down approach while another part has bottom-up approach.
- This integration testing cannot be used for smaller system with huge interdependence between different modules.

Validation Testing

Validation Testing, carried out by QA professionals, is to determine if the system complies with the requirements and performs functions for which it is intended and meets the organization's goals and user needs. This kind of testing is very important, as well as verification testing. Validation is done at the end of the development process and takes place after verification is completed.

Thus, to ensure customer satisfaction, developers apply validation testing. Its goal is to validate and be confident about the product or system and that it fulfils the requirements given by the customer. The acceptance of the software from the end customer is also its part.

When software is tested, the motive is to check the quality regarding the found defects and bugs. When defects and bugs are detected, developers fix them. After that, the software is checked again to make sure no bugs are left. In that way, the software product's quality scales up.

The aim of software testing is to measure the quality of software in terms of a number of defects found in it, the number of tests run and the system covered by the tests. When bugs or defects are found with the help of testing, the bugs are logged and the development team fixes them. Once the bugs are fixed, testing is carried out again to ensure that they are indeed fixed and no new

defects have been introduced in the software. With the entire cycle, the quality of the software increases.

Stages of Validation testing Process:

- Validation Planning – To plan all the activities that need to be included while testing.
- Define Requirements – To set goals and define the requirements for testing.
- Selecting a Team – To select a skilled and knowledgeable development team (the third party included).
- Developing Documents – To develop a user specification document describing the operating conditions.
- Estimation/Evaluation – To evaluate the software as per the specifications and submit a validation report.
- Fixing bugs or Incorporating Changes – To change the software so as to remove any errors found during evaluation.

Validation Testing Variations:

- Component/Unit Testing – The aim of the unit testing is to look for bugs in the software component. At the same time, it also verifies the work of modules and objects which can be tested separately.
- Integration testing- This is an important part of the software validation model, where the interaction between the different interfaces of the components is tested. Along with the interaction between the different parts of the system, the interaction of the system with the computer operating system, file system, hardware, and any other software system it might interact with, is also tested.
- System testing- System testing is carried out when the entire software system is ready. The main concern of system testing is to verify the system against the specified requirements. While carrying out the tests, the tester is not concerned with the internals of the system but checks if the system behaves as per expectations.
- Acceptance testing- During this testing, a tester literally has to think like the client and test the software with respect to user needs, requirements, business processes and determine whether the software can be handed over to the client or not.
- Alpha testing- This type of testing is done at the developers' site by potential customers/users. Any problems encountered during this testing are rectified by the developers then and there.
- Beta testing- Once the software passes the alpha testing stage, beta testing is done at the user's end.
- Regression testing- This testing is done after the desired changes or modifications are made to the existing code. The code, when put to test, may have certain errors that can be resolved by making essential changes. The software is again put to test after these changes are made to check whether the new code fulfils customer requirements or not.

There is a notion as Independent Validation testing – If the validation tests are carried out by a third party, they are known as independent validation and verification (IV&V). The developer needs to provide the user manual to the third party tester. This manual should clearly contain the standard working conditions of the software. These third-party organizations submit a validation

report to the developer after the software is tested. The developer, upon receipt of this report, makes the required changes to the software and repeats tests it to check whether the customer needs are met or not.

Software validation testing is an important part of the software development lifecycle (SDLC), apart from verification, debugging, and certification. Validation testing ensures that the software meets the quality standards set by the customer and that the product meets customer requirements.

System Testing

System Testing is the testing of a complete and fully integrated software product. Usually, software is only one element of a larger computer-based system. Ultimately, software is interfaced with other software/hardware systems. System Testing is actually a series of different tests whose sole purpose is to exercise the full computer-based system.

Two Category of Software Testing

- Black Box Testing
- White Box Testing

System test falls under the black box testing category of software testing.

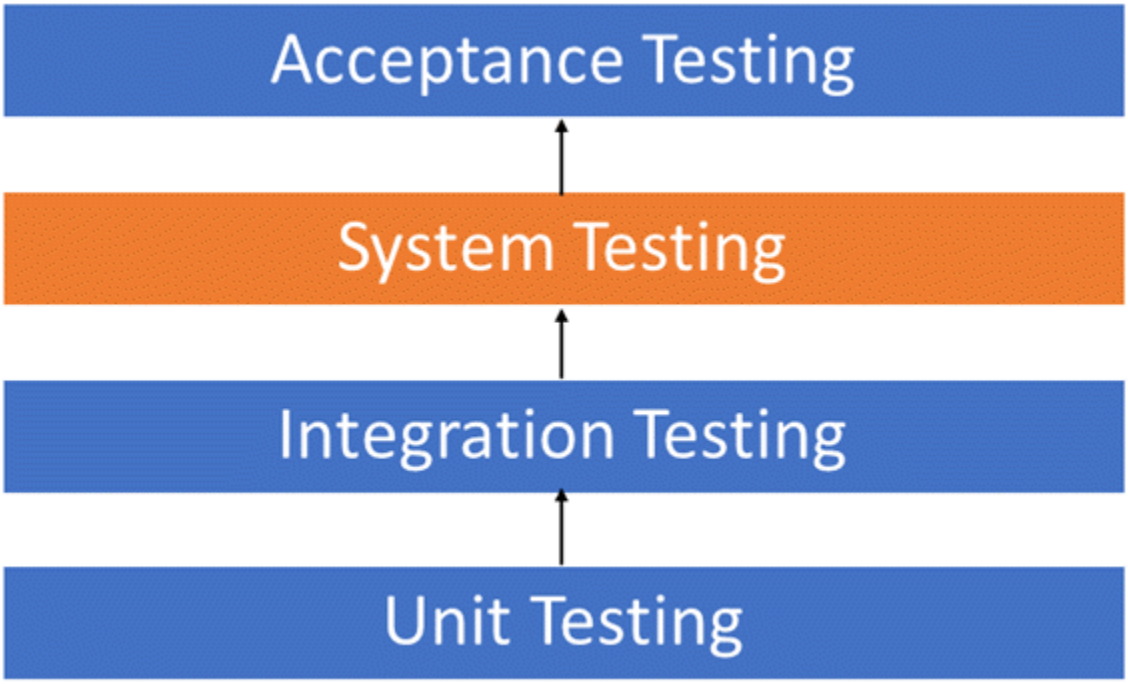
White box testing is the testing of the internal workings or code of a software application. In contrast, black box or System Testing is the opposite. System test involves the external workings of the software from the user's perspective.

System Testing involves testing the software code for following

- Testing the fully integrated applications including external peripherals in order to check how components interact with one another and with the system as a whole. This is also called End to End testing scenario.
- Verify thorough testing of every input in the application to check for desired outputs.
- Testing of the user's experience with the application.

That is a very basic description of what is involved in system testing. You need to build detailed test cases and test suites that test each aspect of the application as seen from the outside without looking at the actual source code.

Software Testing Hierarchy



As with almost any software engineering process, software testing has a prescribed order in which things should be done. The following is a list of software testing categories arranged in chronological order. These are the steps taken to fully test new software in preparation for marketing it:

- Unit testing - testing performed on each module or block of code during development. Unit Testing is normally done by the programmer who writes the code.

- Integration testing - testing done before, during and after integration of a new module into the main software package. This involves testing of each individual code module. One piece of software can contain several modules which are often created by several different programmers. It is crucial to test each module's effect on the entire program model.

- System testing - testing done by a professional testing agent on the completed software product before it is introduced to the market.

- Acceptance testing - beta testing of the product done by the actual end users.

Different Types of System Testing

There are more than 50 types of System Testing. For an exhaustive list of software testing types click here. Below we have listed types of system testing a large software development company would typically use

1. Usability Testing - Usability Testing mainly focuses on the user's ease to use the application, flexibility in handling controls and ability of the system to meet its objectives

2. Load Testing - Load Testing is necessary to know that a software solution will perform under real-life loads.

3. Regression Testing- - Regression Testing involves testing done to make sure none of the changes made over the course of the development process have caused new bugs. It also makes sure no old bugs appear from the addition of new software modules over time.

4. Recovery Testing - Recovery testing is done to demonstrate a software solution is reliable, trustworthy and can successfully recoup from possible crashes.

5. Migration Testing - Migration testing is done to ensure that the software can be moved from older system infrastructures to current system infrastructures without any issues.

6. Functional Testing - Also known as functional completeness testing, Functional Testing involves trying to think of any possible missing functions. Testers might make a list of additional functionalities that a product could have to improve it during functional testing.

7. Hardware/Software Testing - IBM refers to Hardware/Software testing as "HW/SW Testing". This is when the tester focuses his/her attention on the interactions between the hardware and software during system testing.

- Testing Budget - Money becomes a factor not just for smaller companies and individual software developers but large companies as well.

Art of Debugging

In the context of software engineering, debugging is the process of fixing a bug in the software. In other words, it refers to identifying, analyzing and removing errors. This activity begins after the software fails to execute properly and concludes by solving the problem and successfully testing the software. It is considered to be an extremely complex and tedious task because errors need to be resolved at all stages of debugging.

Debugging Process: Steps involved in debugging are:

- Problem identification and report preparation.
- Assigning the report to software engineer to the defect to verify that it is genuine.
- Defect Analysis using modeling, documentations, finding and testing candidate flaws, etc.
- Defect Resolution by making required changes to the system.
- Validation of corrections.

Debugging Strategies:

1. Study the system for the larger duration in order to understand the system. It helps debugger to construct different representations of systems to be debugging depends on the need. Study of the system is also done actively to find recent changes made to the software.
2. Backwards analysis of the problem which involves tracing the program backward from the location of failure message in order to identify the region of faulty code. A detailed study of the region is conducting to find the cause of defects.
3. Forward analysis of the program involves tracing the program forwards using breakpoints or print statements at different points in the program and studying the results. The region where the wrong outputs are obtained is the region that needs to be focused to find the defect.
4. Using the past experience of the software debug the software with similar problems in nature. The success of this approach depends on the expertise of the debugger.

Debugging Tools:

Debugging tool is a computer program that is used to test and debug other programs. A lot of public domain software like gdb and dbx are available for debugging. They offer console-based command line interfaces. Examples of automated debugging tools include code based tracers, profilers, interpreters, etc.

Some of the widely used debuggers are:

- Radare2

- WinDbg
- Valgrind

Difference Between Debugging and Testing:

Debugging is different from testing. Testing focuses on finding bugs, errors, etc whereas debugging starts after a bug has been identified in the software. Testing is used to ensure that the program is correct and it was supposed to do with a certain minimum success rate. Testing can be manual or automated. There are several different types of testing like unit testing, integration testing, alpha and beta testing, etc.

Debugging requires a lot of knowledge, skills, and expertise. It can be supported by some automated tools available but is more of a manual process as every bug is different and requires a different technique, unlike a pre-defined testing mechanism.

Software Testing Fundamentals

Software Testing is evaluation of the software against requirements gathered from users and system specifications. Testing is conducted at the phase level in software development life cycle or at module level in program code. Software testing comprises of Validation and Verification.

Software Validation

Validation is process of examining whether or not the software satisfies the user requirements. It is carried out at the end of the SDLC. If the software matches requirements for which it was made, it is validated.

- Validation ensures the product under development is as per the user requirements.
- Validation answers the question – "Are we developing the product which attempts all that user needs from this software ?".
- Validation emphasizes on user requirements.

Software Verification

Verification is the process of confirming if the software is meeting the business requirements, and is developed adhering to the proper specifications and methodologies.

- Verification ensures the product being developed is according to design specifications.
- Verification answers the question– "Are we developing this product by firmly following all design specifications ?"
- Verifications concentrates on the design and system specifications.

Target of the test are -

- Errors - These are actual coding mistakes made by developers. In addition, there is a difference in output of software and desired output, is considered as an error.
- Fault - When error exists fault occurs. A fault, also known as a bug, is a result of an error which can cause system to fail.

- Failure - failure is said to be the inability of the system to perform the desired task. Failure occurs when fault exists in the system.

Manual Vs Automated Testing

Testing can either be done manually or using an automated testing tool:

- Manual - This testing is performed without taking help of automated testing tools. The software tester prepares test cases for different sections and levels of the code, executes the tests and reports the result to the manager.

  Manual testing is time and resource consuming. The tester needs to confirm whether or not right test cases are used. Major portion of testing involves manual testing.

- Automated This testing is a testing procedure done with aid of automated testing tools. The limitations with manual testing can be overcome using automated test tools.

A test needs to check if a webpage can be opened in Internet Explorer. This can be easily done with manual testing. But to check if the web-server can take the load of 1 million users, it is quite impossible to test manually.

There are software and hardware tools which helps tester in conducting load testing, stress testing, regression testing.

Testing Approaches

Tests can be conducted based on two approaches –

- Functionality testing
- Implementation testing

When functionality is being tested without taking the actual implementation in concern it is known as black-box testing. The other side is known as white-box testing where not only functionality is tested but the way it is implemented is also analyzed.

Exhaustive tests are the best-desired method for a perfect testing. Every single possible value in the range of the input and output values is tested. It is not possible to test each and every value in real world scenario if the range of values is large.

Black-box testing

It is carried out to test functionality of the program. It is also called ‘Behavioral’ testing. The tester in this case, has a set of input values and respective desired results. On providing input, if the output matches with the desired results, the program is tested ‘ok’, and problematic otherwise.

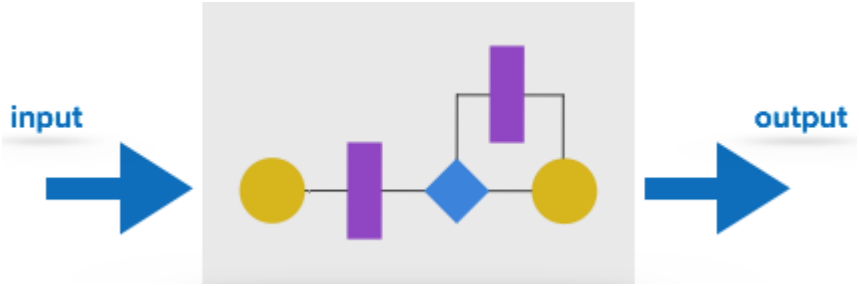
In this testing method, the design and structure of the code are not known to the tester, and testing engineers and end users conduct this test on the software.

Black-box testing techniques:

- Equivalence class - The input is divided into similar classes. If one element of a class passes the test, it is assumed that all the class is passed.
- Boundary values - The input is divided into higher and lower end values. If these values pass the test, it is assumed that all values in between may pass too.
- Cause-effect graphing - In both previous methods, only one input value at a time is tested. Cause (input) – Effect (output) is a testing technique where combinations of input values are tested in a systematic way.
- Pair-wise Testing - The behavior of software depends on multiple parameters. In pairwise testing, the multiple parameters are tested pair-wise for their different values.
- State-based testing - The system changes state on provision of input. These systems are tested based on their states and input.

White-box testing

It is conducted to test program and its implementation, in order to improve code efficiency or structure. It is also known as ‘Structural’ testing.



In this testing method, the design and structure of the code are known to the tester. Programmers of the code conduct this test on the code.

The below are some White-box testing techniques:

- Control-flow testing - The purpose of the control-flow testing to set up test cases which covers all statements and branch conditions. The branch conditions are tested for both being true and false, so that all statements can be covered.

- Data-flow testing - This testing technique emphasis to cover all the data variables included in the program. It tests where the variables were declared and defined and where they were used or changed.

Testing Levels

Testing itself may be defined at various levels of SDLC. The testing process runs parallel to software development. Before jumping on the next stage, a stage is tested, validated and verified.

Testing separately is done just to make sure that there are no hidden bugs or issues left in the software. Software is tested on various levels -

Unit Testing

While coding, the programmer performs some tests on that unit of program to know if it is error free. Testing is performed under white-box testing approach. Unit testing helps developers decide that individual units of the program are working as per requirement and are error free.

Integration Testing

Even if the units of software are working fine individually, there is a need to find out if the units if integrated together would also work without errors. For example, argument passing and data updation etc.

System Testing

The software is compiled as product and then it is tested as a whole. This can be accomplished using one or more of the following tests:

- Functionality testing - Tests all functionalities of the software against the requirement.
- Performance testing - This test proves how efficient the software is. It tests the effectiveness and average time taken by the software to do desired task. Performance testing is done by means of load testing and stress testing where the software is put under high user and data load under various environment conditions.
- Security & Portability - These tests are done when the software is meant to work on various platforms and accessed by number of persons.

Acceptance Testing

When the software is ready to hand over to the customer it has to go through last phase of testing where it is tested for user-interaction and response. This is important because even if the software matches all user requirements and if user does not like the way it appears or works, it may be rejected.

- Alpha testing - The team of developer themselves perform alpha testing by using the system as if it is being used in work environment. They try to find out how user would react to some action in software and how the system should respond to inputs.
- Beta testing - After the software is tested internally, it is handed over to the users to use it under their production environment only for testing purpose. This is not as yet the

delivered product. Developers expect that users at this stage will bring minute problems, which were skipped to attend.

Regression Testing

Whenever a software product is updated with new code, feature or functionality, it is tested thoroughly to detect if there is any negative impact of the added code. This is known as regression testing.

Testing Documentation

Testing documents are prepared at different stages -

Before Testing

Testing starts with test cases generation. Following documents are needed for reference –

- SRS document - Functional Requirements document
- Test Policy document - This describes how far testing should take place before releasing the product.
- Test Strategy document - This mentions detail aspects of test team, responsibility matrix and rights/responsibility of test manager and test engineer.
- Traceability Matrix document - This is SDLC document, which is related to requirement gathering process. As new requirements come, they are added to this matrix. These matrices help testers know the source of requirement. They can be traced forward and backward.

While Being Tested

The following documents may be required while testing is started and is being done:

- Test Case document - This document contains list of tests required to be conducted. It includes Unit test plan, Integration test plan, System test plan and Acceptance test plan.
- Test description - This document is a detailed description of all test cases and procedures to execute them.
- Test case report - This document contains test case report as a result of the test.
- Test logs - This document contains test logs for every test case report.

After Testing

The following documents may be generated after testing :

- Test summary - This test summary is collective analysis of all test reports and logs. It summarizes and concludes if the software is ready to be launched. The software is released under version control system if it is ready to launch.

Testing vs. Quality Control, Quality Assurance and Audit

We need to understand that software testing is different from software quality assurance, software quality control and software auditing.

- Software quality assurance - These are software development process monitoring means, by which it is assured that all the measures are taken as per the standards of organization. This monitoring is done to make sure that proper software development methods were followed.
- Software quality control - This is a system to maintain the quality of software product. It may include functional and non-functional aspects of software product, which enhance the goodwill of the organization. This system makes sure that the customer is receiving quality product for their requirement and the product certified as 'fit for use'.
- Software audit - This is a review of procedure used by the organization to develop the software. A team of auditors, independent of development team examines the software process, procedure, requirements and other aspects of SDLC. The purpose of software audit is to check that software and its development process, both conform standards, rules and regulations.

Black-Box Testing

Black box testing is defined as a testing technique in which functionality of the Application Under Test (AUT) is tested without looking at the internal code structure, implementation details and knowledge of internal paths of the software. This type of testing is based entirely on software requirements and specifications.

In BlackBox Testing we just focus on inputs and output of the software system without bothering about internal knowledge of the software program.

The above Black-Box can be any software system you want to test. For Example, an operating system like Windows, a website like Google, a database like Oracle or even your own custom application. Under Black Box Testing, you can test these applications by just focusing on the inputs and outputs without knowing their internal code implementation. Consider the following video tutorial-

How to do BlackBox Testing

Here are the generic steps followed to carry out any type of Black Box Testing.

- Initially, the requirements and specifications of the system are examined.
- Tester chooses valid inputs (positive test scenario) to check whether SUT processes them correctly. Also, some invalid inputs (negative test scenario) are chosen to verify that the SUT is able to detect them.
- Tester determines expected outputs for all those inputs.
- Software tester constructs test cases with the selected inputs.
- The test cases are executed.
- Software tester compares the actual outputs with the expected outputs.
- Defects if any are fixed and re-tested.

Types of Black Box Testing

There are many types of Black Box Testing but the following are the prominent ones -

- Functional testing - This black box testing type is related to the functional requirements of a system; it is done by software testers.
- Non-functional testing - This type of black box testing is not related to testing of specific functionality, but non-functional requirements such as performance, scalability, usability.
- Regression testing - Regression Testing is done after code fixes, upgrades or any other system maintenance to check the new code has not affected the existing code.

Tools used for Black Box Testing:

Tools used for Black box testing largely depends on the type of black box testing you are doing.

- For Functional/ Regression Tests you can use - QTP, Selenium
- For Non-Functional Tests, you can use - LoadRunner, Jmeter

Black Box Testing Techniques

Following are the prominent Test Strategy amongst the many used in Black box Testing

- Equivalence Class Testing: It is used to minimize the number of possible test cases to an optimum level while maintains reasonable test coverage.
- Boundary Value Testing: Boundary value testing is focused on the values at boundaries. This technique determines whether a certain range of values are acceptable by the system or not. It is very useful in reducing the number of test cases. It is most suitable for the systems where an input is within certain ranges.
- Decision Table Testing: A decision table puts causes and their effects in a matrix. There is a unique combination in each column.

Comparison of Black Box and White Box Testing:

Which to choose ???

Black Box Testing	White Box Testing
the main focus of black box testing is on the validation of your functional requirements.	White Box Testing (Unit Testing) validates internal structure and working of your software code
Black box testing gives abstraction from code and focuses on testing effort on the software system behavior.	To conduct White Box Testing, knowledge of underlying programming language is essential. Current day software systems use a variety of programming languages and technologies and its not possible to know all of them.
Black box testing facilitates testing communication amongst modules	White box testing does not facilitate testing communication amongst modules

Black Box Testing and Software Development Life Cycle (SDLC)

Black box testing has its own life cycle called Software Testing Life Cycle (STLC) and it is relative to every stage of Software Development Life Cycle of Software Engineering.

- Requirement - This is the initial stage of SDLC and in this stage, a requirement is gathered. Software testers also take part in this stage.
- Test Planning & Analysis - Testing Types applicable to the project are determined. A Test Plan is created which determines possible project risks and their mitigation.
- Design - In this stage Test cases/scripts are created on the basis of software requirement documents

- Test Execution- In this stage Test Cases prepared are executed. Bugs if any are fixed and re-tested.

White-Box Testing

White Box Testing is defined as the testing of a software solution's internal structure, design, and coding. In this type of testing, the code is visible to the tester. It focuses primarily on verifying the flow of inputs and outputs through the application, improving design and usability, strengthening security. White box testing is also known as Clear Box testing, Open Box testing, Structural testing, Transparent Box testing, Code-Based testing, and Glass Box testing. It is usually performed by developers.

It is one of two parts of the **"Box Testing" approach** to software testing. Its counterpart, **Blackbox testing**, involves testing from an external or end-user type perspective. On the other hand, Whitebox testing is based on the inner workings of an application and revolves around internal testing.

The term "WhiteBox" was used because of the see-through box concept. The clear box or WhiteBox name symbolizes the ability to see through the software's outer shell (or "box") into its inner workings. Likewise, the "black box" in "Black Box Testing" symbolizes not being able to see the inner workings of the software so that only the end-user experience can be tested.

In this tutorial, you will learn-

What do you verify in White Box Testing?

White box testing involves the testing of the software code for the following:

- Internal security holes
- Broken or poorly structured paths in the coding processes
- The flow of specific inputs through the code
- Expected output
- The functionality of conditional loops
- Testing of each statement, object, and function on an individual basis

The testing can be done at system, integration and unit levels of software development. One of the basic goals of whitebox testing is to verify a working flow for an application. It involves testing a series of predefined inputs against expected or desired outputs so that when a specific input does not result in the expected output, you have encountered a bug.

How do you perform White Box Testing?

To give you a simplified explanation of white box testing, we have divided it into **two basic steps**. This is what testers do when testing an application using the white box testing technique:

STEP 1) UNDERSTAND THE SOURCE CODE

The first thing a tester will often do is learn and understand the source code of the application. Since white box testing involves the testing of the inner workings of an application, the tester must be very knowledgeable in the programming languages used in the applications they are testing. Also, the testing person must be highly aware of secure coding practices. Security is often one of the primary objectives of testing software. The tester should be able to find security issues and prevent attacks from hackers and naive users who might inject malicious code into the application either knowingly or unknowingly.

Step 2) CREATE TEST CASES AND EXECUTE

The second basic step to white box testing involves testing the application's source code for proper flow and structure. One way is by writing more code to test the application's source code. The tester will develop little tests for each process or series of processes in the application. This method requires that the tester must have intimate knowledge of the code and is often done by the developer. Other methods include Manual Testing, trial, and error testing and the use of testing tools as we will explain further on in this article.

WhiteBox Testing Example

Consider the following piece of code

```
Printme (int a, int b) {                    ----------  Printme is a function
int result = a+ b;
   If (result> 0)
            Print ("Positive", result)
   Else
            Print ("Negative", result)
   }                                 ----------   End of the source code
```

The goal of WhiteBox testing is to verify all the decision branches, loops, statements in the code.

To exercise the statements in the above code, WhiteBox test cases would be

- A = 1, B = 1
- A = -1, B = -3

White Box Testing Techniques

A major White box testing technique is Code Coverage analysis. Code Coverage analysis eliminates gaps in a Test Case suite. It identifies areas of a program that are not exercised by a

set of test cases. Once gaps are identified, you create test cases to verify untested parts of the code, thereby increasing the quality of the software product

There are automated tools available to perform Code coverage analysis. Below are a few coverage analysis techniques

Statement Coverage:- This technique requires every possible statement in the code to be tested at least once during the testing process of software engineering.

Branch Coverage - This technique checks every possible path (if-else and other conditional loops) of a software application.

Apart from above, there are numerous coverage types such as Condition Coverage, Multiple Condition Coverage, Path Coverage, Function Coverage etc. Each technique has its own merits and attempts to test (cover) all parts of software code. Using Statement and Branch coverage you generally attain 80-90% code coverage which is sufficient.

To learn more in detail refer this article https://www.guru99.com/code-coverage.html

Types of White Box Testing

White box testing encompasses several testing types used to evaluate the usability of an application, block of code or specific software package. There are listed below --

- **Unit Testing:** It is often the first type of testing done on an application. Unit Testing is performed on each unit or block of code as it is developed. Unit Testing is essentially done by the programmer. As a software developer, you develop a few lines of code, a single function or an object and test it to make sure it works before continuing Unit Testing helps identify a majority of bugs, early in the software development lifecycle. Bugs identified in this stage are cheaper and easy to fix.
- **Testing for Memory Leaks**: Memory leaks are leading causes of slower running applications. A QA specialist who is experienced at detecting memory leaks is essential in cases where you have a slow running software application.

Apart from above, a few testing types are part of both black box and white box testing. They are listed as below

- **White Box Penetration Testing:** In this testing, the tester/developer has full information of the application's source code, detailed network information, IP addresses involved and all server information the application runs on. The aim is to attack the code from several angles to expose security threats
- **White Box Mutation Testing**: Mutation testing is often used to discover the best coding techniques to use for expanding a software solution.

Advantages of White Box Testing

- Code optimization by finding hidden errors.
- White box tests cases can be easily automated.
- Testing is more thorough as all code paths are usually covered.
- Testing can start early in SDLC even if GUI is not available.

Disadvantages of WhiteBox Testing

- White box testing can be quite complex and expensive.
- Developers who usually execute white box test cases detest it. The white box testing by developers is not detailed can lead to production errors.
- White box testing requires professional resources, with a detailed understanding of programming and implementation.
- White-box testing is time-consuming, bigger programming applications take the time to test fully.

Metrics for Source Code

Source lines of code (**SLOC**), also known as **lines of code** (**LOC**), is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code. SLOC is typically used to predict the amount of effort that will be required to develop a program, as well as to estimate programming productivity or maintainability once the software is produced.

Measurement methods[edit]

Many useful comparisons involve only the order of magnitude of lines of code in a project. Using lines of code to compare a 10,000-line project to a 100,000-line project is far more useful than when comparing a 20,000-line project with a 21,000-line project. While it is debatable exactly how to measure lines of code, discrepancies of an order of magnitude can be clear indicators of software complexity or man-hours.

There are two major types of SLOC measures: physical SLOC (LOC) and logical SLOC (LLOC). Specific definitions of these two measures vary, but the most common definition of physical SLOC is a count of lines in the text of the program's source code excluding comment lines.[1]

Logical SLOC attempts to measure the number of executable "statements", but their specific definitions are tied to specific computer languages (one simple logical SLOC measure for C-like programming languages is the number of statement-terminating semicolons). It is much easier to create tools that measure physical SLOC, and physical SLOC definitions are easier to explain. However, physical SLOC measures are sensitive to logically irrelevant formatting and style conventions, while logical SLOC is less sensitive to formatting and style conventions. However, SLOC measures are often stated without giving their definition, and logical SLOC can often be significantly different from physical SLOC.

Consider this snippet of C code as an example of the ambiguity encountered when determining SLOC:

```
for (i=0; i<100; i++) printf("hello"); /* How many lines of code is this? */
```

In this example we have:

- 1 physical line of code (LOC),
- 2 logical lines of code (LLOC) (for statement and printf statement),
- 1 comment line.

Depending on the programmer and coding standards, the above "line of code" could be written on many separate lines:

```
/* Now how many lines of code is this? */
for (i=0; i<100; i++)
{
printf("hello");
}
```

In this example we have:

- 4 physical lines of code (LOC): is placing braces work to be estimated?
- 2 logical lines of code (LLOC): what about all the work writing non-statement lines?
- 1 comment line: tools must account for all code and comments regardless of comment placement.

Even the "logical" and "physical" SLOC values can have a large number of varying definitions. Robert E. Park (while at the Software Engineering Institute) and others developed a framework for defining SLOC values, to enable people to carefully explain and define the SLOC measure used in a project. For example, most software systems reuse code, and determining which (if any) reused code to include is important when reporting a measure.

SLOC measures are somewhat controversial, particularly in the way that they are sometimes misused. Experiments have repeatedly confirmed that effort is highly correlated with SLOC[*citation needed*], that is, programs with larger SLOC values take more time to develop. Thus, SLOC can be effective in estimating effort. However, functionality is less well correlated with SLOC: skilled developers may be able to develop the same functionality with far less code, so one program with fewer SLOC may exhibit more functionality than another similar program. Counting SLOC as productivity measure has its caveats, since a developer can develop only a few lines and yet be far more productive in terms of functionality than a developer who ends up creating more lines (and generally spending more effort). Good developers may merge multiple code modules into a single module, improving the system yet appearing to have negative productivity because they remove code. Also, especially skilled developers tend to be assigned the most difficult tasks, and thus may sometimes appear less "productive" than other developers on a task by this measure. Furthermore, inexperienced developers often resort to code duplication, which is highly discouraged as it is more bug-prone and costly to maintain, but it results in higher SLOC.

SLOC counting exhibits further accuracy issues at comparing programs written in different languages unless adjustment factors are applied to normalize languages. Various computer languages balance brevity and clarity in different ways; as an extreme example, most assembly languages would require hundreds of lines of code to perform the same task as a few characters in APL. The following example shows a comparison of a "hello world" program written in C, and the same program written in COBOL - a language known for being particularly verbose.

C	COBOL

# include <stdio.h> 	
 	
intmain() {
printf("\nHello world\n");
}
	identification division.
	program-id. hello .
	procedure division.
	display "hello world"
	goback .
	end program hello .

Lines of code: 4 (excluding whitespace)	Lines of code: 6 (excluding whitespace)

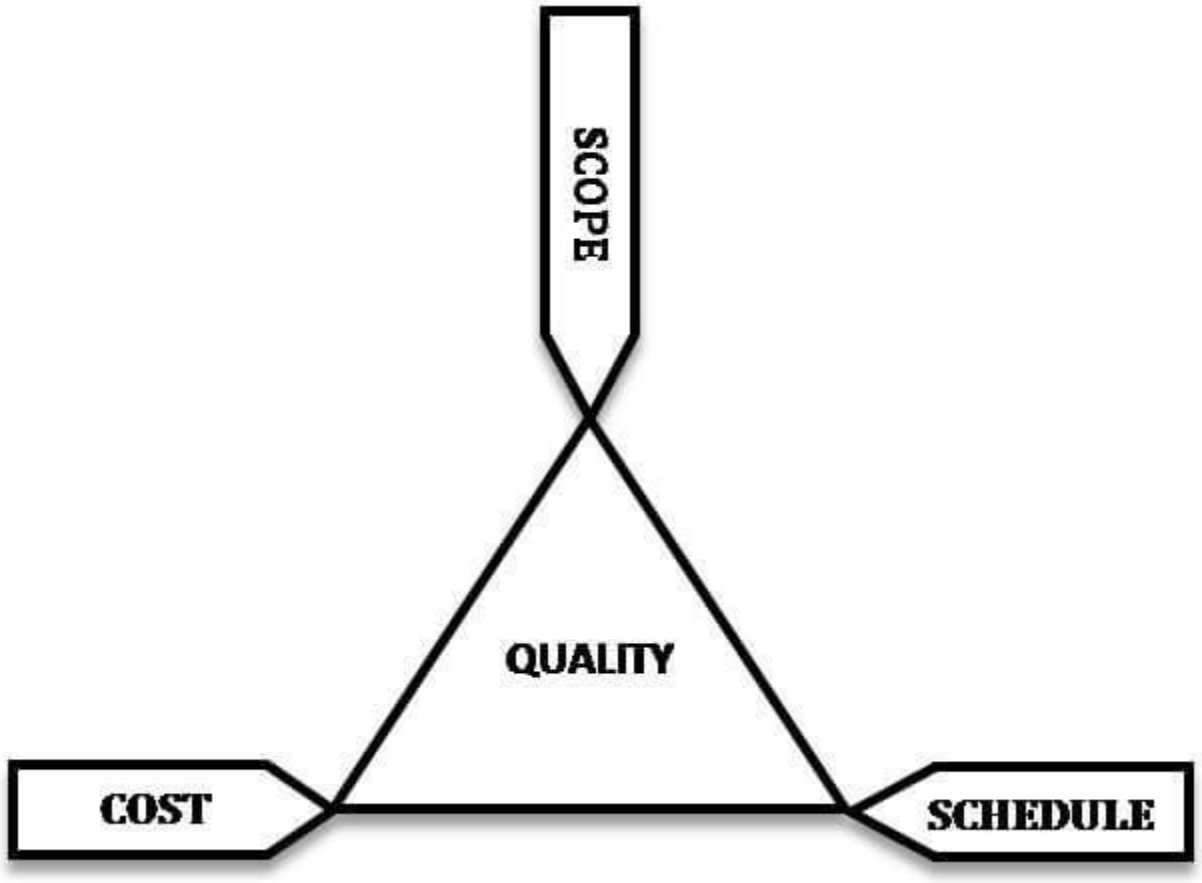
Another increasingly common problem in comparing SLOC metrics is the difference between auto-generated and hand-written code. Modern software tools often have the capability to auto-generate enormous amounts of code with a few clicks of a mouse. For instance, graphical user interface builders automatically generate all the source code for a graphical control elements simply by dragging an icon onto a workspace. The work involved in creating this code cannot reasonably be compared to the work necessary to write a device driver, for instance. By the same token, a hand-coded custom GUI class could easily be more demanding than a simple device driver; hence the shortcoming of this metric.

There are several cost, schedule, and effort estimation models which use SLOC as an input parameter, including the widely used Constructive Cost Model (COCOMO) series of models by Barry Boehm et al., PRICE Systems True S and Galorath's SEER-SEM. While these models have shown good predictive power, they are only as good as the estimates (particularly the SLOC estimates) fed to them. Many[2] have advocated the use of function points instead of SLOC as a measure of functionality, but since function points are highly correlated to SLOC (and cannot be automatically measured) this is not a universally held view.

Metrics for Testing & Maintenance

Software Testing Metric is be defined as a quantitative measure that helps to estimate the progress, quality, and health of a software testing effort. A Metric defines in quantitative terms the degree to which a system, system component, or process possesses a given attribute.

The ideal example to understand metrics would be a weekly mileage of a car compared to its ideal mileage recommended by the manufacturer.



Software testing metrics - Improves the efficiency and effectiveness of a software testing process.

Software testing metrics or software test measurement is the quantitative indication of extent, capacity, dimension, amount or size of some attribute of a process or product.

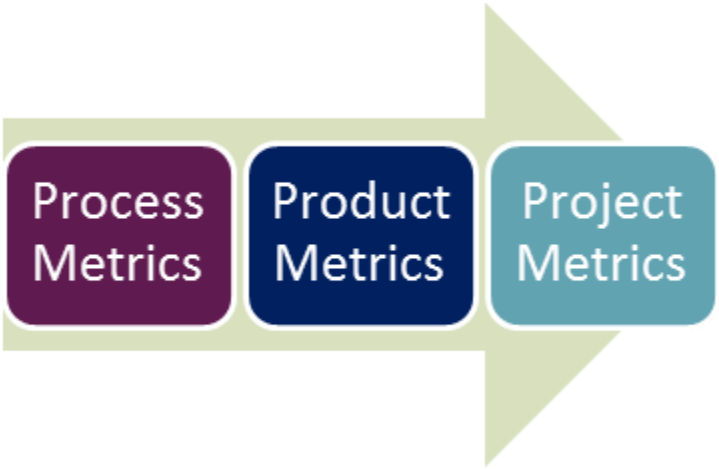
Example for software test measurement: Total number of defects

Why Test Metrics are Important?

"We cannot improve what we cannot measure" and Test Metrics helps us to do exactly the same.

- Take decision for next phase of activities
- Evidence of the claim or prediction
- Understand the type of improvement required
- Take decision or process or technology change

Types of Test Metrics



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

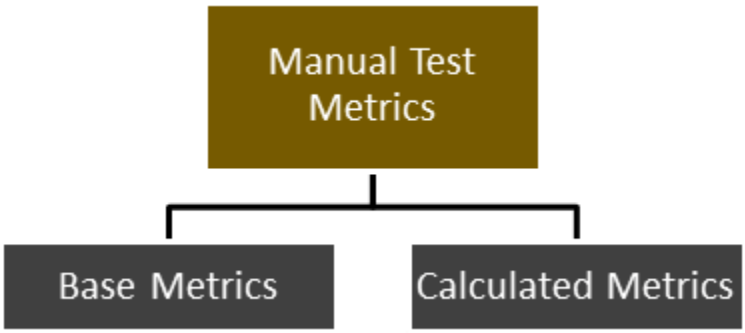
Identification of correct testing metrics is very important. Few things need to be considered before identifying the test metrics

- Fix the target audience for the metric preparation
- Define the goal for metrics
- Introduce all the relevant metrics based on project needs
- Analyze the cost benefits aspect of each metrics and the project lifestyle phase in which it results in the maximum output

Manual Test Metrics

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

- Base Metrics
- Calculated Metrics



Base metrics is the raw data collected by Test Analyst during the test case development and execution (# of test cases executed, # of test cases). While calculated metrics are derived from the data collected in base metrics. Calculated metrics is usually followed by the test manager for test reporting purpose (% Complete, % Test Coverage).

Depending on the project or business model some of the important metrics are

- Test case execution productivity metrics
- Test case preparation productivity metrics
- Defect metrics
- Defects by priority
- Defects by severity
- Defect slippage ratio

Test Metrics Life Cycle

Different stages of Metrics life cycle	Steps during each stage
• Analysis	• Identification of the Metrics • Define the identified QA Metrics
• Communicate	• Explain the need for metric to stakeholder and testing team • Educate the testing team about the data points to need to be captured for processing the metric
• Evaluation	• Capture and verify the data • Calculating the metrics value using the data captured

• Report	• Develop the report with an effective conclusion • Distribute the report to the stakeholder and respective representative • Take feedback from stakeholder

How to calculate Test Metric

Sr#	Steps to test metrics	Example
1	Identify the key software testing processes to be measured	• Testing progress tracking process
2	In this Step, the tester uses the data as a baseline to define the metrics	• The number of test cases planned to be executed per day
3	Determination of the information to be followed, a frequency of tracking and the person responsible	• The actual test execution per day will be captured by the test manager at the end of the day
4	Effective calculation, management, and interpretation of the defined metrics	• The actual test cases executed per day
5	Identify the areas of improvement depending on the interpretation of defined metrics	• The Test Case execution falls below the goal set, we need to investigate the reason and suggest the improvement measures

Example of Test Metric

To understand how to calculate the test metrics, we will see an example of a percentage test case executed.

To obtain the execution status of the test cases in percentage, we use the formula.

Percentage test cases executed= (No of test cases executed/ Total no of test cases written) X 100

Likewise, you can calculate for other parameters like test cases not executed, test cases passed, test cases failed, test cases blocked, etc.

Test Metrics Glossary

- Rework Effort Ratio = (Actual rework efforts spent in that phase/ total actual efforts spent in that phase) X 100
- Requirement Creep = (Total number of requirements added/No of initial requirements)X100
- Schedule Variance = (Actual efforts – estimated efforts) / Estimated Efforts) X 100
- Cost of finding a defect in testing = (Total effort spent on testing/ defects found in testing)
- Schedule slippage = (Actual end date – Estimated end date) / (Planned End Date – Planned Start Date) X 100
- Passed Test Cases Percentage = (Number of Passed Tests/Total number of tests executed) X 100
- Failed Test Cases Percentage = (Number of Failed Tests/Total number of tests executed) X 100
- Blocked Test Cases Percentage = (Number of Blocked Tests/Total number of tests executed) X 100
- Fixed Defects Percentage = (Defects Fixed/Defects Reported) X 100
- Accepted Defects Percentage = (Defects Accepted as Valid by Dev Team /Total Defects Reported) X 100
- Defects Deferred Percentage = (Defects deferred for future releases /Total Defects Reported) X 100
- Critical Defects Percentage = (Critical Defects / Total Defects Reported) X 100
- Average time for a development team to repair defects = (Total time taken for bugfixes/Number of bugs)
- Number of tests run per time period = Number of tests run/Total time
- Test design efficiency = Number of tests designed /Total time
- Test review efficiency = Number of tests reviewed /Total time
- Bug find rote or Number of defects per test hour = Total number of defects/Total number of test hours

Metrics for Software Maintenance

When development of a software product is complete and it is released to the market, it enters the maintenance phase of its life cycle. During this phase the defect arrivals by time interval and customer problem calls (which may or may not be defects) by time interval are the de facto metrics. However, the number of defect or problem arrivals is largely determined by the development process before the maintenance phase. Not much can be done to alter the quality of

the product during this phase. Therefore, these two de facto metrics, although important, do not reflect the quality of software maintenance. What can be done during the maintenance phase is to fix the defects as soon as possible and with excellent fix quality. Such actions, although still not able to improve the defect rate of the product, can improve customer satisfaction to a large extent. The following metrics are therefore very important:

- Fix backlog and backlog management index
- Fix response time and fix responsiveness
- Percent delinquent fixes
- Fix quality

4.3.1 Fix Backlog and Backlog Management Index

Fix backlog is a workload statement for software maintenance. It is related to both the rate of defect arrivals and the rate at which fixes for reported problems become available. It is a simple count of reported problems that remain at the end of each month or each week. Using it in the format of a trend chart, this metric can provide meaningful information for managing the maintenance process. Another metric to manage the backlog of open, unresolved, problems is the backlog management index (BMI).

$$\text{BMI} = \frac{\text{Number of problems closed during the month}}{\text{Number of problem arrivals during the month}} \times 100\%$$

As a ratio of number of closed, or solved, problems to number of problem arrivals during the month, if BMI is larger than 100, it means the backlog is reduced. If BMI is less than 100, then the backlog increased. With enough data points, the techniques of control charting can be used to calculate the backlog management capability of the maintenance process. More investigation and analysis should be triggered when the value of BMI exceeds the control limits. Of course, the goal is always to strive for a BMI larger than 100. A BMI trend chart or control chart should be examined together with trend charts of defect arrivals, defects fixed (closed), and the number of problems in the backlog.

Fix Response Time and Fix Responsiveness

For many software development organizations, guidelines are established on the time limit within which the fixes should be available for the reported defects. Usually the criteria are set in accordance with the severity of the problems. For the critical situations in which the customers' businesses are at risk due to defects in the software product, software developers or the software

change teams work around the clock to fix the problems. For less severe defects for which circumventions are available, the required fix response time is more relaxed. The fix response time metric is usually calculated as follows for all problems as well as by severity level:

Mean time of all problems from open to closed

If there are data points with extreme values, medians should be used instead of mean. Such cases could occur for less severe problems for which customers may be satisfied with the circumvention and didn't demand a fix. Therefore, the problem may remain open for a long time in the tracking report.

In general, short fix response time leads to customer satisfaction. However, there is a subtle difference between fix responsiveness and short fix response time. From the customer's perspective, the use of averages may mask individual differences. The important elements of fix responsiveness are customer expectations, the agreed-to fix time, and the ability to meet one's commitment to the customer.

Percent Delinquent Fixes

The mean (or median) response time metric is a central tendency measure. A more sensitive metric is the percentage of delinquent fixes. For each fix, if the turnaround time greatly exceeds the required response time, then it is classified as delinquent:

This metric, however, is not a metric for real-time delinquent management because it is for closed problems only. Problems that are still open must be factored into the calculation for a real-time metric. Assuming the time unit is 1 week, we propose that the percent delinquent of problems in the active backlog be used. *Active backlog* refers to all opened problems for the week, which is the sum of the existing backlog at the beginning of the week and new problem arrivals during the week. In other words, it contains the total number of problems to be processed for the week—the total workload. The number of delinquent problems is checked at the end of the week. Figure 4.6 shows the real-time delivery index diagrammatically.

Fix Quality

Fix quality or the number of defective fixes is another important quality metric for the maintenance phase. From the customer's perspective, it is bad enough to encounter functional

defects when running a business on the software. It is even worse if the fixes turn out to be defective. A fix is defective if it did not fix the reported problem, or if it fixed the original problem but injected a new defect. For mission-critical software, defective fixes are detrimental to customer satisfaction.

The metric of percent defective fixes is simply the percentage of all fixes in a time interval (e.g., 1 month) that are defective. A defective fix can be recorded in two ways: Record it in the month it was discovered or record it in the month the fix was delivered. The first is a customer measure, the second is a process measure. The difference between the two dates is the latent period of the defective fix. It is meaningful to keep track of the latency data and other information such as the number of customers who were affected by the defective fix. Usually the longer the latency, the more customers are affected because there is more time for customers to apply that defective fix to their software system.