Manual Software Testing

An Excellent Learning Resource

for the Beginners

The term software engineering

Dennis [Dennis 1975]:

Software engineering is the application of principles, skills, and art to the design and construction of programs and systems of programs.

Pomberger and Blaschek [Pomberger 1996]:

Software engineering is the practical application of scientific knowledge for the economical production and use of high-quality software.

Software is:

Instructions (computer programs) which when executed, provide desired function and performance.

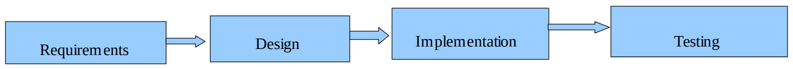
What are software characteristics?

- ⊄ Software is logical unlike hardware, which is physical (contains chips, circuit boards, power supplies etc.,) Hence its characteristics are entirely different.
- ⊄ Software does not “wear out” …as do hardware components, from dust, abuse, temperature and other environmental factors.
- ⊄ A software component should be built such that it can be reused in many different programs

What is the 'software Development life cycle'?

Software life cycle models describe phases of the software cycle and the order in which those phases are executed.There are tons of models, and many companies adopt their own, but all have very similar patterns.The general, basic model is shown below:

General Life Cycle Model



Each phase produces deliverables required by the next phase in the life cycle.Requirements are translated into design.Code is produced during implementation that is driven by the design.Testing verifies the deliverable of the implementation phase against requirements.

They are:

Requirements: Business requirements are gathered in this phase. Meetings with managers, stake holders and users are held in order to determine the requirements.Who is going to use the system?How will they use the system?What data should be input into the system? What data should be output by the system?how the user interface should work?The overall result is the system as a whole and how it performs, not how it is actually going to do it.

Design:The software system design is produced from the results of the requirements phase.This is where the details on how the system will work is produced.Design focuses on high level design like, what programs are needed and how are they going to interact, low-level design (how the individual programs are going to work), interface design (what are the interfaces going to look like) and data design (what data will be required). During these phases, the software's overall structure is defined. Analysis and Design are very crucial in the whole development cycle. Any glitch in the design phase could be very expensive to solve in the later stage of the software development.

Coding: Code is produced from the deliverables of the design phase during implementation, and this is the longest phase of the software development life cycle.For a developer, this is the main focus of the life cycle because this is where the code is produced.Different high level programming languages like C, C++, Pascal, Java are used for coding. With respect to the type of application, the right programming language is chosen.Implementation my overlap with both the design and testing phases.

Testing:During testing, the implementation is tested against the requirements to make sure that the product is actually solving the needs addressed and gathered during the requirements phase.Normally programs are written as a series of individual modules, the system is tested to ensure that interfaces between modules work (integration testing), the system works on the intended platform and with the expected volume of data (volume testing) and that the system does what the user requires (acceptance/beta testing)

Installation, Operation and maintenance: Software will definitely undergo change once it is delivered to the customer. There are many reasons for the change. Change could happen because of some unexpected input values into the system. In addition, the changes in the system could directly affect the software operations.

Water Fall Method:

This is the most common and classic of life cycle models, also referred to as a linear-sequential life cycle model.It is very simple to understand and use.In a waterfall model, each phase must be completed in its entirety before the next phase can begin.At the end of each phase, a review takes place to determine if the project is on the right path and whether or not to continue or discard the project.Unlike what I mentioned in the general model, phases do not overlap in a waterfall model.

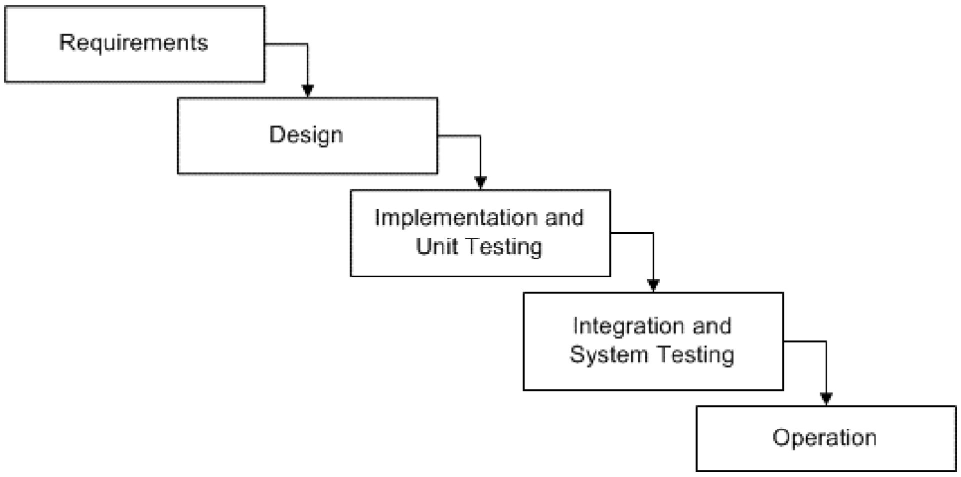
Advantages:

Simple and easy to use.
Easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
Phases are processed and completed one at a time.
Works well for smaller projects where requirements are very well understood.

Disadvantages:

1. Time consuming,
2. Never backward,

3. Difficulty responding to change while developing.
4. No early prototypes of the software are produced.


Requirements
Design
Implementation and Unit Testing
Integration and System Testing
Operation

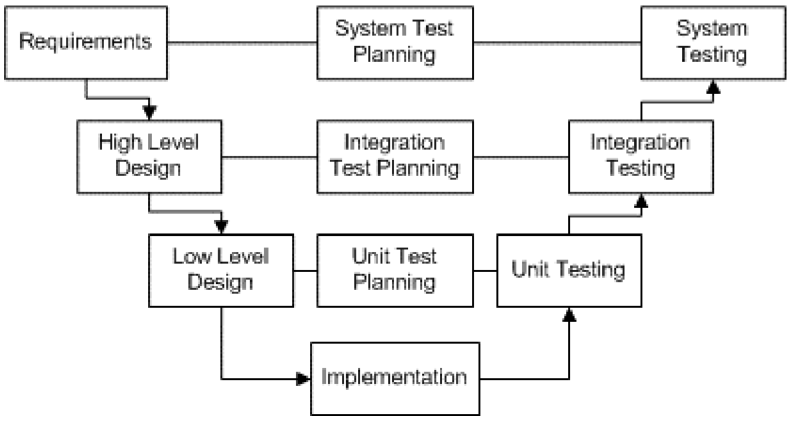
V- model: Just like the waterfall model, the V-Shaped life cycle is a sequential path of execution of processes.Each phase must be completed before the next phase begins.Testing is emphasized in this model more so than the waterfall model though.The testing procedures are developed early in the life cycle before any coding is done, during each of the phases preceding implementation.

Requirements begin the life cycle model just like the waterfall model.Before development is started, a system test plan is created.The test plan focuses on meeting the functionality specified in the requirements gathering.

The high-level design phase focuses on system architecture and design.An integration test plan is created in this phase as well in order to test the pieces of the software systems ability to work together.

The low-level design phase is where the actual software components are designed, and unit tests are created in this phase as well.

The implementation phase is, again, where all coding takes place.Once coding is complete, the path of execution continues up the right side of the V where the test plans developed earlier are now put to use.



Advantages:

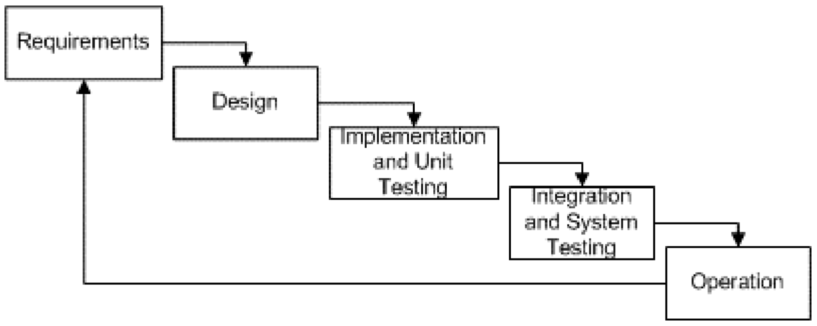
Simple and easy to use.

Each phase has specific deliverables.
Higher chance of success over the waterfall model due to the development of test plans early on during the life cycle.
Works well for small projects where requirements are easily understood.

Disadvantages:

Very rigid, like the waterfall model.
Little flexibility and adjusting scope is difficult and expensive.
Software is developed during the implementation phase, so no early prototypes of the software are produced.

Incremental Model: The incremental model is an intuitive approach to the waterfall model. Multiple development cycles take place here, making the life cycle a "multi-waterfall" cycle. Cycles are divided up into smaller, more easily managed iterations. Each iteration passes through the requirements, design, implementation and testing phases.
A working version of software is produced during the first iteration, so you have working software early on during the software life cycle. Subsequent iterations build on the initial software produced during the first iteration


Requirements
Design
Implementation and Unit Testing
Integration and System Testing
Operation

Advantages

Generates working software quickly and early during the software life cycle.
More flexible – less costly to change scope and requirements.
Easier to manage risk because risky pieces are identified and handled during its iteration.
Each iteration is an easily managed milestone.

Disadvantages

Each phase of an iteration is rigid and do not overlap each other.
Problems may arise pertaining to system architecture because not all requirements are gathered up front for the entire software life cycle.

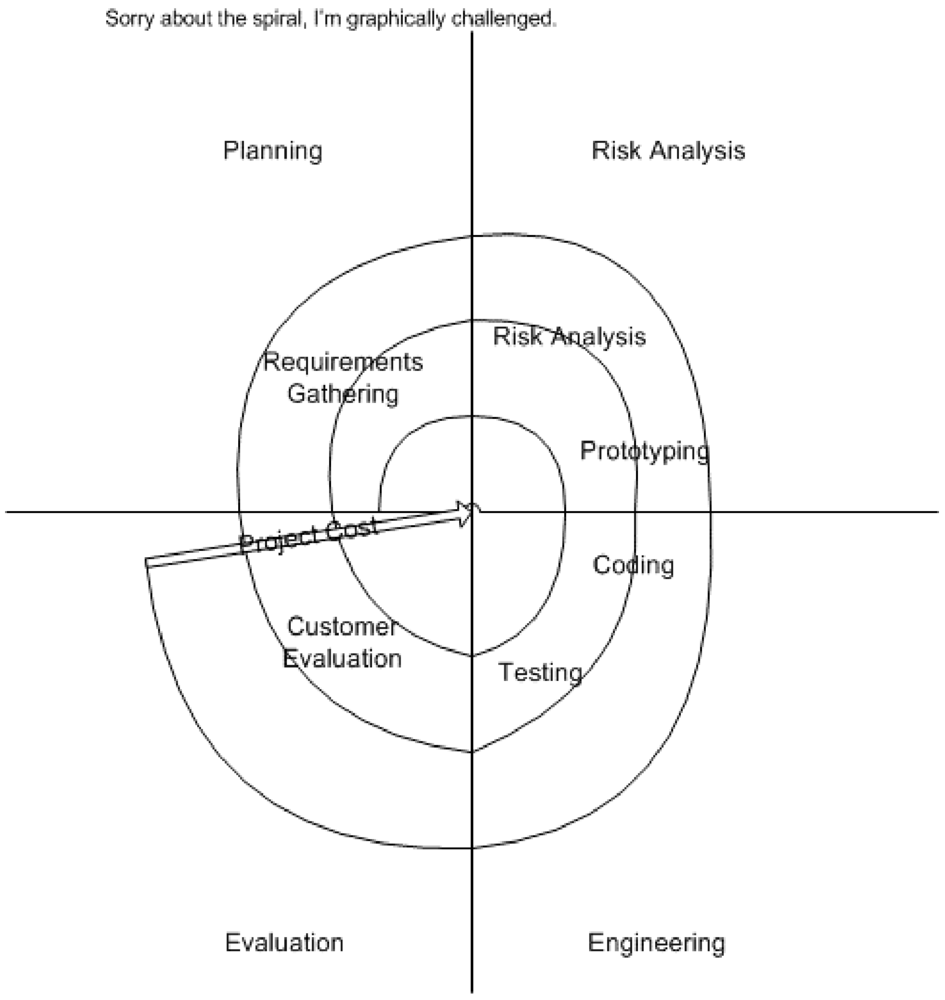
The spiral model:

The spiral model is similar to the incremental model, with more emphases placed on risk analysis.The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation.A software project repeatedly passes through these phases in iterations (called Spirals in this model).The baseline spiral, starting in the planning phase, requirements are gathered and risk is assessed.Each subsequent spirals builds on the baseline spiral.

Requirements are gathered during the planning phase.In the risk analysis phase, a process is undertaken to identify risk and alternate solutions.A prototype is produced at the end of the risk analysis phase.

Software is produced in the engineering phase, along with testing at the end of the phase.The evaluation phase allows the customer to evaluate the output of the project to date before the project continues to the next spiral.

In the spiral model, the angular component represents progress, and the radius of the spiral represents cost.The spiral model is intended for large, expensive, and complicated projects.



Advantages

High amount of risk analysis
Good for large and mission-critical projects.
Software is produced early in the software life cycle.

Disadvantages

Can be a costly model to use.
Risk analysis requires highly specific expertise.
Project's success is highly dependent on the risk analysis phase.
Doesn't work well for smaller projects. What are the four components of the Software Development Process?

a. Plan

b. Do

c. Check

d. Act

What are common problems in the software development process?

1.**Poor requirements** :If requirements are unclear, incomplete, too general, or not testable, there will be problems.

2.**Unrealistic Schedule**:If too much work is crammed in too little time, problems are inevitable

3.**Inadequate testing**:No one will know whether or not the program is any good until the customer complains or systems crash.

4.**Features**:A request to pile on new features after development is underway; extremely common.

5.**Miscommunication**:If developers don't know what's needed or customer's have erroneous expectations, problems are guaranteed

(or)

Reasons for defect injections

1. Incorrect estimations – If effort is not estimated properly, defect injection rate will increase.
2. Vague requirements - if requirements are unclear, incomplete, too general, and not testable, there will be problems.
3. Changes after development - requests to pile on new features after development is underway; extremely common.
4. Miscommunication - if developers don't know what's needed or customers' have erroneous expectations, problems are guaranteed.

Life Cycle Testing means perform testing in parallel with systems development

Life Cycle testing-Role of Testers

- ∉ **Concept Phase**

Evaluate Concept Document,

Learn as much as possible about the product and project

Analyze Hardware/software Requirements,

Strategic Planning

- ∉ **Requirement Phase**

Analyze the Requirements

Verify the Requirements using the review methods.

Prepare Test Plan

Identify and develop requirement based test cases

- ∉ **Design Phase,**

Analyze design specifications

Verify design specifications

Identify and develop Function based test cases

Begin performing Usability Tests

- ∉ **Coding Phase**

Analyze the code Verify

the code

Code Coverage

Unit test

- ∉ **Integration & Test Phase**

Integration Test

Function Test

System Test

Performance Test

Review user manuals

- ∉ **Operation/Maintenance Phase**

Monitor Acceptance Test

Develop new validation tests for confirming problems

Software TESTING :

The process of executing computer software in order to determine whether the results it produces are correct", Glass '79
"The process of executing a program with the intent of finding errors", Myers '79
A *"DESTRUCTIVE"*, yet creative process
Testing is the measure of software quality"

Testing is the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements, (or) to identify differences between

expected and actual results."(or)Software testing is an important technique for assessing the quality of a software product. A purpose of testing is
to cause failures in order to make faults visible [10] so that the faults can be fixed and not be delivered in the code that goes to customers. Program testing can be used to show the *presence* of bugs, but never to show their absence.

1.Testing is the process of showing the presence of defects.

2.There is no absolute notion of \correctness".

3.Testing remains the most cost effective approach to building confidence within most software systems.

Test plan:

A **test plan** is a systematic approach to testing a system such as a machine or software. The plan typically contains a detailed understanding of what the eventual work flow will be (or)

The ***test plan*** defines the objectives and scope of the testing effort, and identifies the methodology that your team will use to conduct tests. It also identifies the hardware, software, and tools required for testing and the features and functions that will be tested. A well-rounded test plan notes any risk factors that jeopardize testing and includes a testing schedule(or)

A test plan is a document describing the scope, approach, resources,and schedule of intended test activities. It identifies test items, the features to be tested,the testing tasks, who will do each task, and any risks requiring contingency plans. An important component of the test plan is the individual test cases. Some form of test plan should be developed prior to any test.

TESTCASE:A test case is a set of test inputs, execution conditions, and expected results to determine if a feature of an application is working correctly, such as to exercise a particular program path or to verify compliance with a specific requirement. Think diabolically! What are the worst things someone could try to do to your program? Write test for these.

Testing process: The test development life cycle contains the following components/steps:

System study-Assess development plan and status

Collect SFS/SRS

Develop Test plan

Design test cases

Tests specifications:This document includes technical details (Software requirements)required prior to the testing.

Execute tests

Evaluate tests

Evaluate results

Acceptance test

Document results(defect reports, summary reports)

Recreating the problem is essentially important in testing so that problems that are identified can be repeated and corrected.

Testing strategy:Test Strategy provides the road map describing the steps to be undertaken while testing, and the effort, time and resources required for the testing.

2.The test Strategy should incorporate test planning, test case design, test execution, resultant data collection and data analysis.

Monkey Testing:

Testing the application randomly like hitting keys irregularly and try to brake down the system there is no specific test cases and scenarios for monkey testing

Verification refers to set of activities that involves reviews and meetings to evaluate documents, plans, code, requirements and specifications(to ensure that software correctly implements a specific function, imposed at the start of that phase); this can be done with checklists, issues lists, and walkthroughs and inspection meetings.

For example, in the software for the Monopoly game, we can verify that two players cannot own the same house.

Validation *is the process of evaluating a system* ***or component during or at the end of the development process to determine whether it*** *satisfies specified requirements. (or)*

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

For example, we validate that when a player lands on "Free Parking," they get all the money that was collected. validation always involves comparison against requirements.

In the language of V&V, black box testing is often used for validation (are we building the right software?) and white box testing is often used for verification (are we building the software right?).

Quality Assurance (QA) The system implemented by an organization which assures outside bodies that the data generated is of proven and known quality and meets the needs of the end user. This assurance relies heavily on **documentation of processes, procedures, capabilities, and monitoring of such.**

- Helps establish processes
- Identifies weaknesses in processes and improves them
- QA is the responsibility of the entire team

Quality Control (QC) Mechanism to ensure that the required quality characteristics exist in the finished product (or)

It is a system of routine technical activities, to measure and control the quality of the product as it is being developed..

Quality control activities may be fully automated, entirely manual, or a combination of automated tools and human interaction

- ∉ implements the process.
- ∉ Identifies defects for the primary purpose of correcting defects.
- ∉ QC is the responsibility of the tester.

White-Box Testing

White-box testing is a verification technique software engineers can use to examine if their code works as expected or

White-box test design allows one to peek inside the "box", and it focuses specifically on using internal knowledge of the software to guide the selection of test data.

The tests written based on the white box testing strategy incorporate coverage of the code written, branches, paths, statements and internal logic of the code etc.Test data required for such testing is less exhaustive than that of Black Box testing.

White-box testing is also known as structural testing, clear box testing, and glass box testing (Beizer, 1995). The connotations of "clear box" and "glass box" appropriately indicate that you have full visibility of the internal workings of the software product, specifically, the logic and the structure of the code**.**

Using the white-box testing techniques outlined in this chapter, a software engineer can design test cases that

(1) exercise independent paths within a module or unit;
(2) exercise logical decisions on both their true and false side;
(3) execute loops at their boundaries and within their operational bounds; and
(4) exercise internal data structures to ensure their validity

White Box Testing Techniques

(a)Basis Path Testing : In most software units, there is a potentially (near) infinite number of different paths through the code, so complete path coverage is impractical

Cyclomatic Complexity :

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of a basis path testing method, the value computed for Cyclomatic complexity defines the number for independent paths in the basis set of a program and provides us an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

White-box testing is used for the following types:

*1**.Unit testing,*** This is done to ensure whether a particular module in a system is working properly.Typically done by the programmer, as it requires detailed knowledge of the internal program design and code. A unit is a software component that cannot be subdivided into other components (IEEE, 1990).Unit testing is important for ensuring the code is solid before it is integrated with other code. **Approximately 65% of all bugs can be** caught in unit testing. May require developing test driver modules or test harnesses.

Testing stubs are frequently used in unit testing. A stub simulates a module which is being called by the program. This is very important because most of the called modules are not ready at this stage.

2.Integration testing, which is testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them (IEEE, 1990). Test cases are written which explicitly examine the interfaces between the various units.

There are two common ways to conduct integration testing.

- Non-incremental Integration Testing
- Incremental Integration Testing

Non-incremental Integration Testing (big bang or umbrella): All the software units are assembled into the entire program. This assembly is then tested as a whole from the beginning, usually resulting in a chaotic situation, as the causes of defects are not easily isolated and corrected

Incremental Integration Testing: The program is constructed and tested in small increments by adding a minimum number of components at each interval. Therefore, the errors are easier to isolate and correct, and the interfaces are more likely to be tested completely.

There are two common approaches to conduct incremental integration testing:

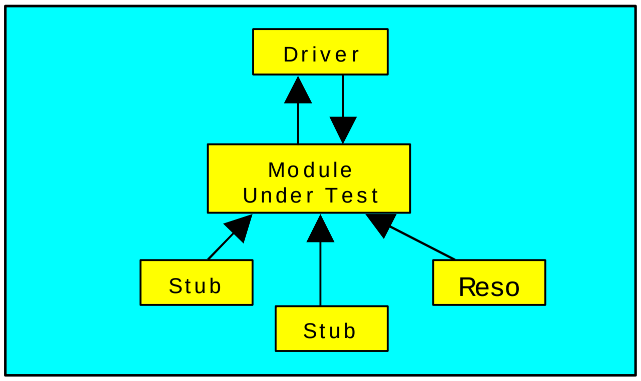
- Top-Down Incremental Integration Testing
- Bottom-up Incremental Integration Testing

(a)Top-Down Incremental Integration Testing: The top-down approach to integration testing requires the highest-level modules be test and integrated first. Modules are integrated from the main module (main program) to the subordinate modules either in the depth-first or breadth-first manner. Integration testing starts with the highest-level, control module, or 'main program', with all its subordinates replaced by stubs.

(b)Bottom-up Incremental Integration Testing: The bottom-up approach requires the lowest-level units be tested and integrated first. The lowest level sub-modules are integrated and tested, then the successively superior level components are added and tested, transiting the hierarchy from the bottom, upwards

Scaffolding is defined as computer programs and data files built to support software development and testing but not intended to be included in the final product. Scaffolding code is code that simulates the functions of components that don't exist yet and allow the program to execute [16]. Scaffolding code involves the creation of stubs and test drivers.

Test driver allows you to call a function and display its return values. **Test drivers** are defined as a software module used to involve a module under test and often, provide test inputs, controls, and monitor execution and report test results [11]. Test drivers simulate the calling components (e.g. hard-coded method calls)



A stub returns a value that is sufficient for testing

Stubs are modules that simulate components that aren't written yet, formally defined as a *computer program statement substituting for the body of a software module that is or will be defined elsewhere* [11]. For example, you might write a skeleton of a method with just the method signature and a hard-coded but valid return value.

Example - For Unit Testing of 'Sales Order Printing' program, a 'Driver' program will have the code which will create Sales Order records using hard coded data and then call 'Sales Order Printing' program. Suppose this printing program uses another unit which calculates Sales discounts by some complex calculations. Then call to this unit will be replaced by a 'Stub', which will simply return fix discount data.

Mock objects are temporary substitutes for domain code that emulates the real code. For example, if the program is to interface with a database, you might not want to wait for the database to be fully designed and created before you write and test a partial program. You can create a mock object of the database that the program can use temporarily. The interface of the mock object and the real object would be the same.

BLACK BOX TESTING

Also known as behavioral, functional, opaque-box, closed-box, concrete box testing. Test engineer need not know the internal working of the "Black box". It focuses on the functionality part of the module. Black box testing is testing against the specification.Test data needs to be exhaustive in such testing.(or)The test designer selects valid and invalid input and determines the correct output. There is no knowledge of the test object's internal structure.While this method can uncover unimplemented parts of the specification, one cannot be sure that all existent paths are tested.

Testing Strategy: The base of the Black box testing strategy lies in the selection of appropriate data as per functionality and testing it against the functional specifications in order to check for normal and abnormal behavior of the system.

Strategies for Black Box Testing

Ideally, we'd like to test every possible thing that can be done with our program. But, as we said, writing and executing test cases is expensive. We want to make sure that we definitely write test cases for the kinds of things that the customer will do most often or even fairly often. Our objective is to find as many defects as possible in as few test cases as possible. To accomplish this objective, we use some strategies that will be discussed in this subsection. We want to avoid writing redundant test cases that won't tell us anything new (because they have similar conditions to other test cases we already wrote). Each test case should probe a different mode of failure. We also want to design the simplest test cases that could possibly reveal this mode of failure – test cases themselves can be error-prone if we don't keep this in mind.

Black Box Testing Techniques:

1.Error Guessing:

2.Equivalence Partitioning:To keep down our testing costs, we don't want to write several test cases that test the same aspect of our program. A good test case uncovers a different class of errors (e.g., incorrect processing of all character data) than has been uncovered by prior test cases. Equivalence partitioning is a strategy that can be used to reduce the number of test cases that need to be developed. Equivalence partitioning divides the input domain of a program into classes. For each of these equivalence classes, the set of data should be treated the same by the module under test and should produce the same answer. Test cases should be designed so the inputs lie within these equivalence classes. Once you have identified these partitions, you choose test cases from each partition. To start, choose *a typical value somewhere in the middle of (or well into) each of these two ranges.*

1.To reduce the number of test cases to a necessary minimum.
2.To select the right test cases to cover all possible scenarios.

An additional effect by applying this technique is that you also find the so called "dirty" test cases

3.Boundary Value Analysis:Boris Beizer, well-known author of testing book advises, “Bugs lurk in corners and congregate at boundaries.” Programmers often make mistakes on the boundaries of the equivalence classes/input domain. As a result, we need to focus testing at these boundaries. This type of testing is called Boundary Value Analysis (BVA) and guides you to create test cases at the “edge” of the equivalence classes. *Boundary value* is defined as a *data value that corresponds to a minimum or maximum*

When creating BVA test cases, consider the following [17]:

1. If input conditions have a range from **a** to **b** (such as a=100 to b=300), create test
cases:

- immediately below **a** (99)
- at **a** (100)

- immediately above **a** (101)
- immediately below **b** (299)
- at **b** (300)
- immediately above **b** (301)

2.If input conditions specify a *number* of values that are allowed, test these limits. For example, input conditions specify that only one train is allowed to start in each direction on each station. In testing, try to add a second train to the same station/same direction. If (somehow) three trains could start on one station/direction, try to add two trains (pass), three trains (pass), and four trains (fail).

Decision Table Testing:

Decision tables are used to record complex business rules that must be implemented in the program, and therefore tested. A sample decision table is found in Table 7. In the table, the conditions represent possible input conditions. The actions are the events that should trigger, depending upon the makeup of the input conditions. Each column in the table is a unique combination of input conditions (and is called a rule) that result in triggering the action(s) associated with the rule. Each rule (or column) should become a test case.

If a Player (A) lands on property owned by another player (B), A must pay rent to B. If A does not have enough money to pay B, A is out of the game.

Decision table*:*

	Rule 1	Rule 2	Rule 3
Conditions			
A lands on B’s property	Yes	Yes	No
A has enough money to pay rent	Yes	No	----
Actions			
A stays in game	Yes	No	Yes

Black Box Testing Types:

(0)Integration testing **–** This involves testing of combined parts of an application to determine if they function together correctly. The 'parts' can be code modules, individual applications, client and server applications on a network, etc. This type of testing is especially relevant to client/server and distributed systems.

(1)Functional Testing:
Function testing is used to find out differences between the program and its external specifications. Testing that ignores the internal mechanism or structure of a system or component and focuses on the outputs generated in response to selected inputs and execution conditions. This stage will include Validation Testing.

(2)Smoke Testing(sanity testing):

Smoke is the initial level of testing effort to determine if the new software version is performing well enough for its major level of testing effort. The Smoke test scenarios should emphasize breadth more than depth. All components should be touched, and every major feature should be tested briefly. This is normally done when a change in program is made and the tester wants to ensure that this change has not impacted the program at some other location. A subset of the regression test cases can be set aside as smoke tests. A *smoke test* is a *group of test cases that establish that the system is stable and all major functionality is present and works under "normal" conditions.* The purpose of smoke tests is to demonstrate stability, not to find bugs with the system. (or)

Sanity Testing is performed to test validate the stability of the build (software application under test). Under sanity testing we check whether the application is readily accessible and the check for basic functionalities. Once this are verified we go ahead with the test case execution. We start when the build released

(3)Exploratory Testing(ad-hoc):
Exploratory Tests are categorized under Black Box Tests and are aimed at testing in conditions when sufficient time is not available for testing or proper documentation is not available. This type of testing is done without any formal Test Plan or Test Case creation..With ad hoc testing, people just start trying anything they can think of without any rational road map through the customer requirements.Requires highly skilled resources who can smell errors.(or)
A type of testing where we explore software, write and execute the test scripts simultaneously.

Exploratory testing is a type of testing where tester does not have specifically planned test cases, but he/she does the testing more with a point-of-view to explore the software features and tries to break it in order to find out unknown bugs.

Exploratory Testing is a learn and work type of testing activity where a tester can at least learn more and understand the software if at all he/she was not able to reveal any potential bug The only limit to the extent to which you can perform exploratory testing is your imagination and creativity, more you can think of ways to explore, understand the software, more test cases you will be able write and execute simultaneously.

For smaller groups or projects, an ad-hoc process is more appropriate

Advantages of Exploratory Testing:

It helps testers in learning new strategies, expand the horizon of their imagination that helps them in understanding & executing more and more test cases and finally improve their productivity. Exploratory testing helps tester in confirming that he/she understands the application and its functionality properly and has no confusion about the working of even a smallest part of it, hence covering the most important part of requirement understanding. As in case of exploratory testing, we write and execute the test cases simultaneously. It helps in collecting result oriented test scripts and shading of load of unnecessary test cases which do not yield and result.

(4)Regression Testing

Rerunning test cases which a program has previously executed correctly in order to detect errors spawned by changes or corrections made during software development and maintenance. Ensures that changes have not propagated unintended side affects.(or)

Regression testing, which is selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements

This is a more detailed kind of testing which starts once the application passes the Smoke testing.

Most of the time the testing team is asked to check last minute changes in the code just before making a release to the client, in this situation the testing team needs to check only the affected areas.

Regression tests are a subset of the original set of test cases. The purpose of running the regression test case is to make a “spot check” to examine whether the new code works properly and has not damaged any previously-working functionality by propagating unintended side effects.

Most often, it is impractical to re-run all the test cases when changes are made. Since regression tests are run throughout the development cycle, there can be white box regression tests at the unit and integration levels and black box tests at the integration, function, system, and acceptance test levels.

The Regression test suit contains different classes of test cases:

*(a)Retesting:*Tests that focus on the software components that have been changed.*(or)*

Retesting

In the Test Execution if we come across any bugs for that will be reported to the Development. Once the bugs are fixed in the subsequent build the bugs that are fixed will be RETESTED. This is Retesting.

REGRESSION TESTING

In the regression testing instead of validating the bugs that were fixed , we validate the impact of bugs on the dependent functionalities.....

(b)Regional Regression Testing:

(c)Full Regression Testing: Tests that will exercise all software

functions

(5)System Testing:
System testing concentrates on testing the complete system with a variety of techniques and methods. This is done to compare the program against its original objectives.
(or),
System testing involves putting the new program in many different environments to ensure the program works in typical customer environments with various versions and types of operating systems
Various type of system testing are:
*(a)Compatibility Testing(portability testing):*Testing whether the system is compatible with other systems with which it should communicate.
when you develop applications on one platform, you need to check if the application works on other operating systems as well. This is the main goal of Compatibility Testing.
(b)Recovery Testing:
Testing aimed at verifying the system's ability to recover from varying degrees of failure. Testing how well a system recovers from crashes, hardware failures, or other catastrophic problems.(or)
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

(c)Security Testing:
Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. During Security testing, password cracking, unauthorized entry into the software, network security are all taken into consideration.It is done to check the security objectives of a program. This testing might not be required in some cases.

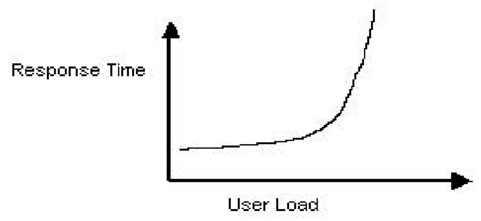
(6)Usability Testing(UI testing):
Usability is the degree to which a user can easily learn and use a product to achieve a goal. A simpler description is testing the software from a user's point of view.This is done to check whether the user interfaces are proper, output of the program is relevant
Are the communication device(s) designed in a manner such that the information is displayed in a understandable fashion enabling the operator to correctly interact with the system?

(7)Performance Testing:
This is done to validate whether the output of a particular program is given within a particular time period. In general, we want to measure the Response Time, Throughput, and Utilization of the Web site while simulating attempts by virtual users to simultaneously access the site. One of the main objectives of performance testing is to maintain a Web site with low response time, high throughput, and low utilization.(or)
T*esting conducted to evaluate the compliance of a system or component with specified performance requirements* . To continue the

above example, a performance requirement might state that the price lookup must complete in less than 1 second. Performance testing evaluates whether the system can look up prices in less than 1 second (even if there are 30 cash registers running simultaneously).

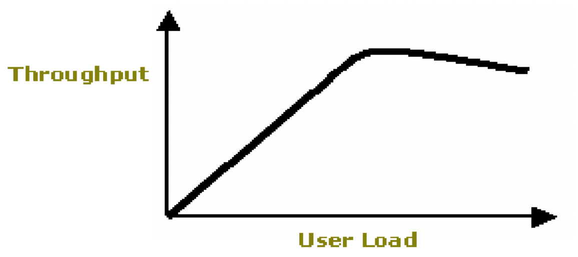
(a)Response Time

Response Time is the delay experienced when a request is made to the server and the server's response to the client is received. It is usually measured in units of time, such as seconds or milliseconds.



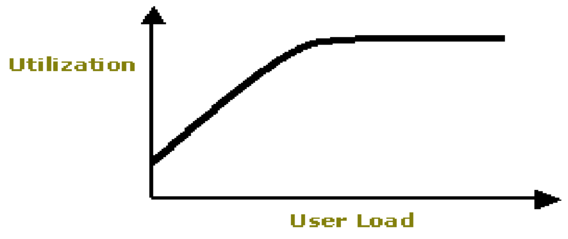
(b)Throughput

Throughput refers to the number of client requests processed within a certain unit of time. Typically, the unit of measurement is requests per second or pages per second. From a marketing perspective, throughput may also be measured in terms of visitors per day or page views per day,



(c)Utilization

Utilization refers to the usage level of different system resources, such as the server's CPU(s), memory, network bandwidth, and so forth. It is usually measured as a percentage of the maximum available level of the specific resource. Utilization versus user load for a Web server typically produces a curve, as shown in Figure



Types of performance testing:

- ⊄ Load testing:Load testing is defined as the testing to determine whether the system is capable ofhandling anticipated number of users or not.

The objective of load testing is to check whether the system can perform well for specified load. The system may be capable of accommodating more than 1000 concurrent users. But, validating that is not under the scope of load testing. No attempt is made to determine how many more concurrent users the system is capable of servicing.

- Stress testing:Unlike load testing where testing is conducted for specified number of users, stress testing is conducted for the number of concurrent users beyond the specified limit. The objective is to identify the maximum number of users the system can handle before breaking down or degrading drastically. Since the aim is to put more stress on system, think time of the user is ignored and the system is exposed to excess load

Let us take the same example of on line shopping application to illustrate the objective of stress testing. It determines the maximum number of concurrent users an on line system can service which can be beyond 1000 users (specified limit). However, there is a possibility that the maximum load that can be handled by the system may found to be same as the anticipated limit.

- Volume Testing:Volume Testing, as its name implies, is testing that purposely subjects a system (both hardware and software) to a series of tests where the volume of data being processed is the subject of the test. Such systems can be transactions processing systems capturing real time sales or could be database updates and or data retrieval.

Volume Testing is to find weaknesses in the system with respect to its handling of large amounts of data during short time periods.This is an expensive type of testing and done only when it is essential.

- Scalability testing

We perform scalability testing to determine how effectively your Web site will expand to accommodate an increasing load. Scalability testing allows you to plan Web site capacity improvements as your business grows and to anticipate problems that could cost you revenue down the line. Scalability testing also reveals when your site cannot maintain good performance at higher usage levels, even with increased capacity.

(8)Installation Testing

Installation testing is often the most under tested area in testing. This type of testing is performed to ensure that all Installed features and options function properly. It is also performed to verify that all necessary components of the application are, indeed, installed. The Uninstallation of the product also needs to be tested to ensure that all data, executables, and DLL files are removed.

(9)Alpha Testing

Alpha testing happens at the development site just before the roll out of the application to the customer. Alpha tests are conducted replicating the live environment where the application would be installed and running. A software prototype stage when the software is first available for run. Here the software has the core functionalities in it but complete functionality is not aimed at. It would be able to accept inputs and give outputs. The test is conducted at the developer's site only.

In a software development cycle, depending on the functionalities the number of alpha phases required is laid down in the project plan itself.

During this, the testing is not a through one, since only the prototype of the software is available. Basic installation – uninstallation tests, the completed core functionalities are tested.

(10)Acceptance testing (UAT or Beta testing) :- also called beta testing, application testing, and end user testing - is a phase of software development in which the software is tested in the "real world" by the intended audience. UAT can be done by in-house testing in which volunteers use the software by making the test version available for downloading and free trial over the Web .The experiences of the early users are forwarded back to the developers who make final changes before releasing the software commercially. It is the formal means by which we ensure that the new system or process does actually meet the essential user requirements.

Acceptance testing allows customers to ensure that the system meets their business requirements. In fact, depending on how your other tests were performed, this final test may not always be necessary. If the customers and users participated in system tests, such as requirements testing and usability testing, they may not need to perform a formal acceptance test. However, this additional test will probably be required for the customer to give final approval for the system.

The actual acceptance test follows the general approach of the Acceptance Test Plan. After the test is completed, the customer either accepts the system or identifies further changes that are required. After these subsequent changes are completed, either the entire test is performed again or just those portions in question are retested.

Results of these tests will allow both the customers and the developers to be confident that the system will work as intended.

11.Robustness testing*:*Test cases are chosen outside the domain to test robustness to unexpected, erroneous input.

12.Defensive testing:Which includes *tests under both normal and abnormal conditions*

13.Soak testing (Endurance testing):

It is performed by running high volume of data for a prolonged period. This is to check the behavior of a system on a busy day. Normally this kind of testing is a must for banking products.

14.End to End Testing - This is the testing of a complete application environment in a situation that simulates actual use, such as interacting with other applications if applicable

However, methods with very straightforward functionality, for example, getter and setter methods, don't need unit tests unless they do their getting and setting in some "interesting" way. A good guideline to follow is to write a unit test whenever you feel the need to comment some behavior in the code. If you're like many programmers who aren't fond of commenting code, unit tests are a way of documenting your code behavior. Avoid using domain objects in unit tests

While writing unit tests, I have used the following guidelines to determine if the unit test being written is actually a functional test:

If a unit test crosses class boundaries, it might be a functional test.
If a unit test is becoming very complicated, it might be a functional test.
If a unit test is fragile (that is, it is a valid test but it has to change continually to handle different user permutations), it might be a functional test.
If a unit test is harder to write than the code it is testing, it might be a functional test.

Conclusion

Unit tests are written from the developer's perspective and focus on particular methods of the class under test. Use these guidelines when writing unit tests:

Write the unit test before writing code for class it tests.
Capture code comments in unit tests.
Test all the public methods that perform an "interesting" function (that is, not getters and setters, unless they do their getting and setting in some unique way).
Put each test case in the same package as the class it's testing to gain access to package and protected members.
Avoid using domain-specific objects in unit tests.

Functional tests are written from the user's perspective and focus on system behavior. After there is a suite of functional tests for the system, the members of the development team responsible for functional testing should bombard the system with variations of the initial tests.

A test case suite is simply a table of contents for the individual test cases

Organizing the suite of test cases by priority, functional area, actor, business object, or release can help identify parts of the system that need additional test cases.

We can test the application for Negative testing by giving the invalid inputs...for example in order to testing pin for ATM it only accepts 4 lettered PIN...try giving 3 lettered or 5-lettered.... all zeroes and so on.......

What is meant by Framework?Is the framework related only to AUTOMATIO or else it will applicable to MANUAL testing too?

a)   Framework is nothing but how to execute the test case process
b)   That is clearly what steps I will follow to the given test cases

(or)

Frame Work means, what type of actions or process we r following before going to test the applications.....in simple words we can say "Architecture".it will be applicable for both, but in terms of terminology we use only for automation.

Whether we write one test case for one use case?

```
Some use cases are very simple so that only one test case
is required to cover the functionality many times use cases
are very complicated so that a lot of test cases are
required to cover the whole functionality
example:"use case->Applicant login the screen
This use case needs a minimum of 10 test cases id to cover
the whole functionality
```

How to Report Bugs Effectively

In a nutshell, the aim of a bug report is to enable the programmer to see the program failing in front of them. You can either show them in person, or give them careful and detailed instructions on how to make it fail. If they can make it fail, they will try to gather extra information until they know the cause. If they can't make it fail, they will have to ask you to gather that information for them.

In bug reports, try to make very clear what are actual facts ("I was at the computer and this happened") and what are speculations ("I *think* the problem might be this"). Leave out speculations if you want to, but don't leave out facts.

They don't know what's happened, and they can't get close enough to watch it happening for themselves, so they are searching for clues that might give it away. Error messages, incomprehensible strings of numbers, and even unexplained delays are all just as important as fingerprints at the scene of a crime. Keep them!

If you manage to get out of the problem, whether by closing down the affected program or by rebooting the computer, a good thing to do is to try to make it happen again. Programmers like problems that they can reproduce more than once. Happy programmers fix bugs faster and more efficiently.

you describe the symptoms, the actual discomforts and aches and pains and rashes and fevers, and you let the doctor do the diagnosis of what the problem is and what to do about it. Otherwise the doctor dismisses you as a hypochondriac or crackpot, and quite rightly so.

It's the same with programmers. Providing your own diagnosis might be helpful sometimes, but always state the symptoms. The diagnosis is an optional extra, and not an alternative to giving the symptoms. Equally, sending a modification to the code to fix the problem is a useful addition to a bug report but not an adequate substitute for one.

If a programmer asks you for extra information, don't make it up! Somebody reported a bug to me once, and I asked him to try a command that I knew wouldn't work. The reason I asked him to try it was that I wanted to know which of two different error messages it would give. Knowing which error message came back would give a vital clue. But he didn't actually try it - he just mailed me back and said "No, that won't work". It took me some time to persuade him to try it for real.

Say "intermittent fault" to any programmer and watch their face fall. The easy problems are the ones where performing a simple sequence of actions will cause the failure to occur. The programmer can then repeat those actions under closely observed test conditions and watch what happens in great detail. Too many problems simply don't work that way: there will be programs that fail once a week, or fail once in a blue moon, or never fail when you try them in front of the programmer but always fail when you have a deadline coming up. Most intermittent faults are not truly intermittent. Most of them have some logic somewhere. Some might occur when the machine is running out of memory, some might occur when another program tries to modify a critical file at the wrong moment, and some might occur only in the first half of every hour! (I've actually seen one of these.)

Also, if you can reproduce the bug but the programmer can't, it could very well be that their computer and your computer are different in some way and this difference is causing the problem. I had a program once whose window curled up into a little ball in the top left corner of the screen, and sat there and *sulked*. But it only did it on 800x600 screens; it was fine on my 1024x768 monitor

Be specific. If you can do the same thing two different ways, state which one you used. "I selected Load" might mean "I clicked on Load" or "I pressed Alt-L". Say which you did. Sometimes it matters.

Be verbose. Give more information rather than less. If you say too much, the programmer can ignore some of it. If you say too little, they have to come back and ask more questions. One bug report I received was a single sentence; every time I asked for more information, the reporter would reply with another single sentence. It took me several weeks to get a useful amount of information, because it turned up one short sentence at a time.
Be careful of pronouns. Don't use words like "it", or references like "the window", when it's unclear what they mean. Consider this: "I started FooApp. It put up a warning window. I tried to close it and it crashed." It isn't clear what the user tried to close. Did they try to close the warning window, or the whole of FooApp? It makes a difference. Instead, you could say "I started FooApp, which put up a warning window. I tried to close the warning window, and FooApp crashed." This is longer and more repetitive, but also clearer and less easy to misunderstand.
Read what you wrote. Read the report back to yourself, and see if *you* think it's clear. If you have listed a sequence of actions which should produce the failure, try following them yourself, to see if you missed a step.

Summary

The first aim of a bug report is to let the programmer see the failure with their own eyes. If you can't be with them to make it fail in front of them, give them detailed instructions so that they can make it fail for themselves.
In case the first aim doesn't succeed, and the programmer *can't* see it failing themselves, the second aim of a bug report is to describe what went wrong. Describe everything in detail. State what you saw, and also state what you expected to see. Write down the error messages, *especially* if they have numbers in.
When your computer does something unexpected, *freeze*. Do nothing until you're calm, and don't do anything that you think might be dangerous.
By all means try to diagnose the fault yourself if you think you can, but if you do, you should still report the symptoms as well.
Be ready to provide extra information if the programmer needs it. If they didn't need it, they wouldn't be asking for it. They aren't being deliberately awkward. Have version numbers at your fingertips, because they will probably be needed.
Write clearly. Say what you mean, and make sure it can't be misinterpreted.
Above all, *be precise*. Programmers like precision.

Don't write titles like these:

1."Can't install" - Why can't you install? What happens when you try to install?
2."Severe Performance Problems" - ...and they occur when you do what?
3."back button does not work" - Ever? At all?

Good bug titles:
1."1.0 upgrade installation fails if Mozilla M18 package present" - Explains problem and the context.
2."RPM 4 installer crashes if launched on Red Hat 6.2 (RPM 3) system" - Explains what happens, and the context.

What are the different types of Bugs we normally see in any of the Project? Include the severity as well.

1. User Interface Defects -------------------------------- Low
2. Boundary Related Defects ------------------------------- Medium
3. Error Handling Defects --------------------------------- Medium
4. Calculation Defects ------------------------------------ High
5. Improper Service Levels (Control flow defects) --------- High

6. Interpreting Data Defects ----------------------------- High
7. Race Conditions (Compatibility and Intersystem defects)- High
8. Load Conditions (Memory Leakages under load) ----------- High
9. Hardware Failures:------------------------------------- High

(or)

1)GUI Related(spelling mistake,non-uniformity of text box, colors)--Low--P3

2)Functionality related--Medium--P2

3)System/application crashes---High---P1

Generally high priority defect are those on basis of those build may be rejected

BUG LIFE CYCLE:

The Life Cycle of a bug in general context is:
So let me explain in terms of a tester's perspective:
A tester finds a new defect/bug, so using a defect tracking tool logs it.

1. Its status is 'NEW' and assigns to the respective Dev team (Team lead orManager).
2. The team lead assigns it to the team member, so the status is 'ASSIGNED TO'
3. The developer works on the bug fixes it and re-assigns to the tester for testing. Now the status is 'RE-ASSIGNED'
4. The tester, check if the defect is fixed, if its fixed he changes the status to 'VERIFIED'
5. If the tester has the authority (depends on the company) he can after verifying change the status to 'FIXED'. If not the test lead can verify it and change the status to 'fixed'.
6. If the defect is not fixed he re-assigns the defect back to the Dev team for re-fixing.

So this is the life cycle of a bug.

Bug status description:
These are various stages of bug life cycle. The status caption may vary depending on the bug tracking system you are using.

1) New: When QA files new bug.

2) Deferred: If the bug is not related to current build or can not be fixed in this release or bug is not important to fix immediately then the project manager can set the bug status as deferred.

3) Assigned: ‘Assigned to’ field is set by project lead or manager and assigns bug to developer.

4) Resolved/Fixed: When developer makes necessary code changes and verifies the changes then he/she can make bug status as ‘Fixed’ and the bug is passed to testing team.

5) Could not reproduce: If developer is not able to reproduce the bug by the steps given in bug report by QA then developer can mark the bug as ‘CNR’. QA needs action to check if bug is reproduced and can assign to developer with detailed reproducing steps.

6) Need more information: If developer is not clear about the bug reproduce steps provided by QA to reproduce the bug, then he/she can mark it as “Need more information’. In this case QA needs to add detailed reproducing steps and assign bug back to dev for fix.

7) Reopen: If QA is not satisfy with the fix and if bug is still reproducible even after fix then QA can mark it as ‘Reopen’ so that developer can take appropriate action.

8) Closed: If bug is verified by the QA team and if the fix is ok and problem is solved then QA can mark bug as ‘Closed’.

9) Rejected/Invalid: Some times developer or team lead can mark the bug as Rejected or invalid if the system is working according to specifications and bug is just due to some misinterpretation.

LIFE CYCLE OF SOFTWARE TESTING:

- Requirement gathering: Includes collecting requirements, functional specifications, and other necessary documents
- Obtain end dates for completion of testing
- Identify resources and their responsibilities, required standards and processes.
- Identify application's high risk area, prioritize testing, and determine scope and assumptions of testing
- Determine test strategy and types of testing to be used – Smoke, System, Performance etc
- Determine hardware and software requirements
- Determine test data requirements
- Divide the testing into various tasks
- Schedule testing process
- Create test cases: A test case is a document that describes an

input, action, or event and an expected response, to determine if a feature of an application is working correctly.

- Review the test cases
- Finalize test environment and test data. Make sure the test environment is separated from development environment.
- Get and install software deliveries (basically the developed application)
- Perform tests using the test cases available
- Analyze and inform results
- The defect needs to be informed and assigned to programmers who can fix those.
- Track reported defects till closure
- Perform another cycle/s if needed : After the defects is fixed, the program needs to be re-tested, for the fix done and also to confirm that defect fixing has not injected some other defect.
- Update test cases, test environment, and test data through life cycle as per the change in requirements, if any.

What is an inspection?

A: An inspection is a formal meeting, more formalized than a walk-through and typically consists of 3-10 people including a moderator, reader (the author of whatever is being reviewed) and a recorder (to make notes in the document). The subject of the inspection is typically a document, such as a requirements document or a test plan. The purpose of an inspection is to find problems and see what is missing, not to fix anything. The result of the meeting should be documented in a written report.

What makes a good test engineer?

A: A person is a good test engineer if he

- Has a "test to break" attitude,
- Takes the point of view of the customer,
- Has a strong desire for quality,
- Has an attention to detail, He's also
- Tactful and diplomatic and
- Has good a communication skill, both oral and written. And he
- Has previous software development experience, too.

Good test engineers have a "test to break" attitude, they take the point of view of the customer, have a strong desire for quality and an attention to detail.

AND

In case of GOOD QUALITY ASSURANCE ENGINEERS (Good QA engineers),they need to understand the entire software development process and how it fits into the business approach and the goals of the organization. Communication skills and the ability to understand various sides of issues are important.

What makes a good QA/Test Manager?

A: QA/Test Managers are familiar with the software development process; able to maintain enthusiasm of their team and promote a positive atmosphere; able to promote teamwork to increase productivity; able to promote cooperation between Software and Test/QA Engineers

How do you know when to stop testing?

A: This can be difficult to determine. Many modern software applications are so complex and run in such an interdependent environment, that complete testing can never be done. Common factors in deciding when to stop are...

- Deadlines, e.g. release deadlines, testing deadlines;
- Test cases completed with certain percentage passed;

- Test budget has been depleted;
- Coverage of code, functionality, or requirements reaches a specified point;
- Bug rate falls below a certain level; or
- Beta or alpha testing period ends.

What if there isn't enough time for thorough testing?

A: Since it's rarely possible to test every possible aspect of an application, every possible combination of events, every dependency, or everything that could go wrong, risk analysis is appropriate to most software development projects. Use risk analysis to determine where testing should be focused. This requires judgment skills, common sense and experience. The checklist should include answers to the following questions:

- Which functionality is most important to the project's intended purpose?
- Which functionality is most visible to the user?
- Which functionality has the largest safety impact?
- Which functionality has the largest financial impact on users?
- Which aspects of the application are most important to the customer?
- Which aspects of the application can be tested early in the development cycle?
- Which parts of the code are most complex and thus most subject to errors?
- Which parts of the application were developed in rush or panic mode?
- Which aspects of similar/related previous projects caused problems?
- Which aspects of similar/related previous projects had large maintenance expenses?
- Which parts of the requirements and design are unclear or poorly thought out?
- What do the developers think are the highest-risk aspects of the application?
- What kinds of problems would cause the worst publicity?
- What kinds of problems would cause the most customer service complaints?
- What kinds of tests could easily cover multiple functionalities?
- Which tests will have the best high-risk-coverage to time-required ratio?

Reliability of a software system is defined as the probability that this system fulfills a function (determined by the specifications) for a specified number of input trials under specified input conditions in a specified time interval (assuming that hardware and input are free of errors).

A software system is robust if the consequences of an error in its operation, in the input, or in the hardware, in relation to a given application, are inversely proportional to the probability of the occurrence of this error in the given application.

- Frequent errors (e.g. erroneous commands, typing errors) must be handled with particular care
- Less frequent errors (e.g. power failure) can be handled more laxly, but still must not lead to irreversible consequences.