**What is software testing?**

Software testing is a process of testing the application / product to make sure it is working as expected. It can be achieved using manual testing OR using automation testing.

Another main intention of software testing is to find defects

**Why we need to do testing**

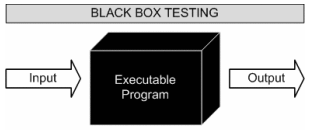
* To prevent defects from end users
* To deliver quality of the product to client
* To maintain confidence with client

**Objectives of testing**

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

**Software Testing Types**

* There are two types of testing
  + - Black Box Testing
    - White Box Testing
* Black Box Testing: Testing team is responsible to do black box testing



Black box testing treats the system as a “black-box”, so it doesn’t explicitly use Knowledge of the internal structure or code. Or in other words the Test engineer need not know the internal working of the “Black box” or application.

Main focus in black box testing is on functionality of the system as a whole.

* White Box Testing:

White box testing requires technical knowledge in which application is developed. In order to do white box testing we need to go through the complete coding means we need to understand each and every line of the code. Most of the time this testing is performed by developers.

**SDLC: Software Development Life Cycle**

Software Development Life Cycle will have below phases and these phases will describe how SDLC works when any new project starting from scratch.

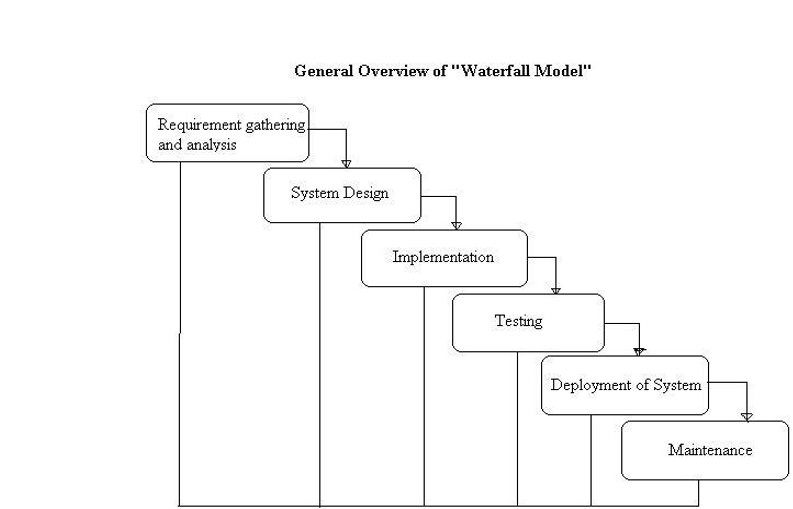
1. Gathering Business Requirements
2. Preparing SRS / BRS document(SRS mean Software Requirements Specification and BRS stands for Business Requirements Specifications)
3. Designing
4. Development / Implementation / Coding
5. Testing
6. Support / Maintenance

**SDLC Models:** We have different types of SDLC models and out of which we mostly used ones are below:

**Waterfall Model:**

The Waterfall Model was first Process Model to be introduced. It is 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 fully before the next phase can begin. This type of model is basically used for the for the project which is small and there are no uncertain requirements. 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. In this model the testing starts only after the development is complete. In **waterfall model phases** do not overlap.

**Diagram of Waterfall-model:**



**Advantages of waterfall model:**

* This model is simple and easy to understand and use.
* It is easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
* In this model phases are processed and completed one at a time. Phases do not overlap.
* Waterfall model works well for smaller projects where requirements are very well understood.

**Disadvantages of waterfall model:**

* Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage.
* No working software is produced until late during the life cycle.
* High amounts of risk and uncertainty.
* Not a good model for complex and object-oriented projects.
* Poor model for long and ongoing projects.
* Not suitable for the projects where requirements are at a moderate to high risk of changing.

**When to use the waterfall model:**

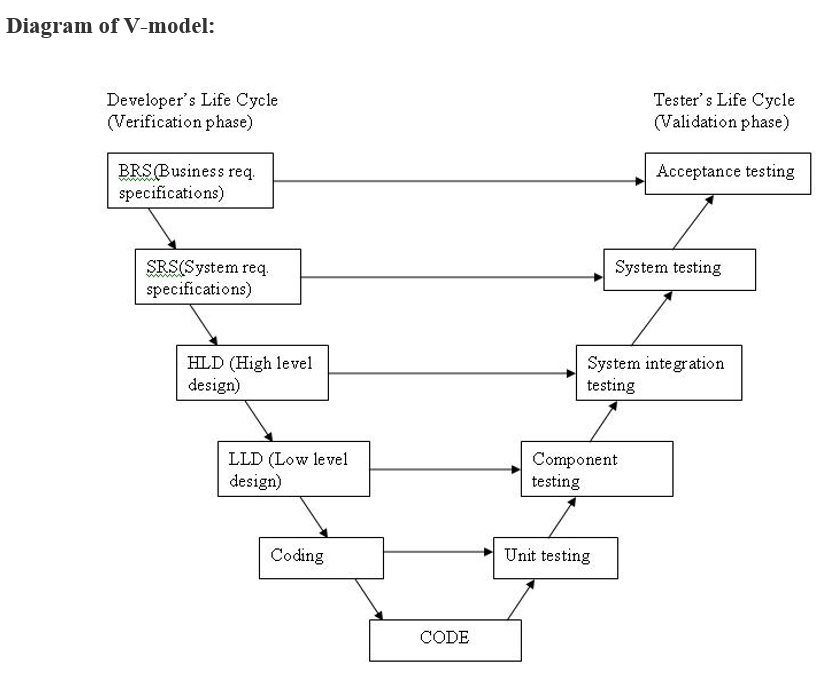
* This model is used only when the requirements are very well known, clear and fixed.
* Product definition is stable.
* There are no ambiguous requirements
* The project is short.

Very less customer enter action is involved during the development of the product. Once the product is ready then only it can be demoed to the end users. Once the product is developed and if any failure occurs then the cost of fixing such issues are very high, because we need to update everywhere from document till the logic

**V Model:**

V- model means Verification and Validation model. Just like the [waterfall model](http://istqbexamcertification.com/what-is-waterfall-model-advantages-disadvantages-and-when-to-use-it/), the V-Shaped life cycle is a sequential path of execution of processes. Each phase must be completed before the next phase begins.  Testing of the product is planned in parallel with a corresponding phase of development.

**Diagram of V-model:**



The various phases of the V-model are as follows:

**Requirements** like BRS and SRS begin the life cycle model just like the waterfall model. But, in this model before development is started, a [system test](http://istqbexamcertification.com/what-is-system-testing/) plan is created.  The test plan focuses on meeting the functionality specified in the requirements gathering.

**The high-level design (HLD)** phase focuses on system architecture and design. It provide overview of solution, platform, system, product and service/process. An integration test plan is created in this phase as well in order to test the pieces of the software systems ability to work together.

**The low-level design** **(LLD)** phase is where the actual software components are designed. It defines the actual logic for each and every component of the system. Class diagram with all the methods and relation between classes comes under LLD. Component tests are created in this phase as well.

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

**Coding:** This is at the bottom of the V-Shape model. Module design is converted into code by developers.

**Advantages of V-model:**

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

**Disadvantages of V-model:**

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

**When to use the V-model:**

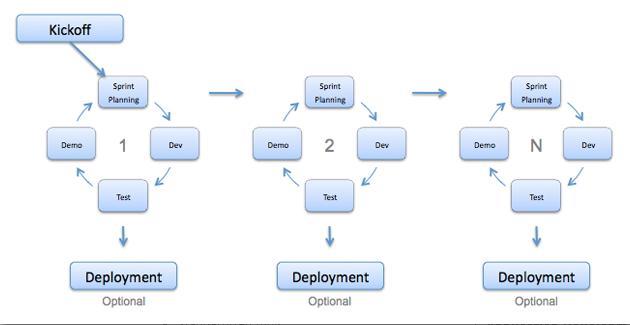
* The V-shaped model should be used for small to medium sized projects where requirements are clearly defined and fixed.
* The V-Shaped model should be chosen when ample technical resources are available with needed technical expertise.

High confidence of customer is required for choosing the V-Shaped model approach. Since, no prototypes are produced, there is a very high risk involved in meeting customer expectations.

**Agile:**

90% of projects are using this model. Agile development model is also a type of Incremental model. Software is developed in incremental, rapid cycles. This results in small incremental releases with each release building on previous functionality. Each release is thoroughly [tested](http://istqbexamcertification.com/why-is-testing-necessary/) to ensure [software quality](http://istqbexamcertification.com/what-is-software-quality/) is maintained. It is used for time critical applications.  Extreme Programming (XP) is currently one of the most well known agile [development life cycle model](http://istqbexamcertification.com/what-are-the-software-development-models/).

**Diagram of Agile model:**



**Advantages of Agile model:**

* Customer satisfaction by rapid, continuous delivery of useful software.
* People and interactions are emphasized rather than process and tools. Customers, developers and testers constantly interact with each other.
* Working software is delivered frequently (weeks rather than months).
* Face-to-face conversation is the best form of communication.
* Close, daily cooperation between business people and developers.
* Continuous attention to technical excellence and good design.
* Regular adaptation to changing circumstances.
* Even late changes in requirements are welcomed

**Disadvantages of Agile model:**

* In case of some software deliverables, especially the large ones, it is difficult to assess the effort required at the beginning of the software development life cycle.
* There is lack of emphasis on necessary designing and documentation.
* The project can easily get taken off track if the customer representative is not clear what final outcome that they want.
* Only senior programmers are capable of taking the kind of decisions required during the development process. Hence it has no place for newbie programmers, unless combined with experienced resources.

**When to use Agile model:**

* When new changes are needed to be implemented. The freedom agile gives to change is very important. New changes can be implemented at very little cost because of the frequency of new increments that are produced.
* To implement a new feature the developers need to lose only the work of a few days, or even only hours, to roll back and implement it.
* Unlike the waterfall model in agile model very limited planning is required to get started with the project. Agile assumes that the end users’ needs are ever changing in a dynamic business and IT world. Changes can be discussed and features can be newly effected or removed based on feedback. This effectively gives the customer the finished system they want or need.
* Both system developers and stakeholders alike, find they also get more freedom of time and options than if the software was developed in a more rigid sequential way. Having options gives them the ability to leave important decisions until more or better data or even entire hosting programs are available; meaning the project can continue to move forward without fear of reaching a sudden standstill.

**Verification and Validation:**

**Verification:** process of evaluating work-products of a development phase to determine whether they meet the specified requirements for that phase. Everyone in project team is responsible for doing verification.

Verification is achieved using below:

* Reviews
* Walkthroughs
* Inspections

**Validation:** process of evaluating software during or at the end of the development process to determine whether it specified requirements. Validation is performed by testing team only.

**Difference between Verification and Validation:**  
  
- Verification is Static Testing whereas Validations is Dynamic Testing.  
- Verification takes place before validation.  
- Verification evaluates plans, document, requirements and specification, where as Validation evaluates product.  
- Verification inputs are checklist, issues list, walkthroughs and inspection ,where as in Validation testing of actual product.  
- Verification output is set of document, plans, specification and requirement documents where as in Validation actual product is output.

**Testing techniques used in software industry:**

In software industry we follow below testing techniques

1. Black box testing techniques
2. White box testing techniques

Let’s try to understand black box testing techniques

Black box testing techniques:

1. Equivalence partitioning
2. Boundary value analysis
3. ErrorGuessing

**Equivalence Partitioning:**

In this method the input domain data is divided into different equivalence data classes. This method is typically used **to reduce the total number of test case**s to a finite set of testable test cases, still covering maximum requirements.

In short it is the process of taking all possible test cases and placing them into classes. One test value is picked from each class while testing.

**E.g.:** If you are testing for an input box accepting numbers from 1 to 1000 then there is no use in writing thousand test cases for all 1000 valid input numbers plus other test cases for invalid data.

Using equivalence partitioning method above test cases can be divided into three sets of input data called as classes. Each test case is a representative of respective class.

So in above example we can divide our test cases into three equivalence classes of some valid and invalid inputs.

EP: Here we divide input into three class

Let’s take an example: A field is taking range of numbers between 1 to 1000.

1. Between the range(valid class)🡪 20
2. Above the range(invalid class)🡪 1002
3. Below the range(invalid class)🡪 -2

**Boundary value analysis:**

It’s widely recognized that input values at the **extreme ends of input** domain cause more errors in system. More application **errors occur at the boundaries** of input domain. ‘Boundary value analysis’ testing technique is used to identify errors at boundaries rather than finding those exist in center of input domain.

Boundary value analysis is a next part of Equivalence partitioning for designing test cases where test cases are selected at the edges of the equivalence classes.

BVA: Here we divide input with respect to boudries of range

Let’s take an example: A field is taking range of numbers between 1 to 1000.

1. N(minmum)🡪 1
2. N-1(minimum-1)🡪 0
3. N (maximum)🡪 1000
4. N+1(maximum+1)🡪1001

0

1

999

1000

1001

**Error Guessing Technique**

**Error guessing is a technique on guessing the error which can prevail in the code**. It is basically an experience based technique where the test analyst uses his / her experience to guess the problematic areas of the application. If the analyst guesses that the login page is error prone, then the testers write more detailed test cases concentrating on the login page. Testers can think of variety of combinations of data to test the login page.

To design test cases based on error guessing technique, Analyst can use the past experiences to identify the conditions. This technique can be used at any level of testing and for testing the common mistakes like:

* Divide by zero
* Entering blank spaces in the text fields
* Pressing submit button without entering values.
* Uploading files exceeding maximum limits.

**Error guessing technique requires skilled and experienced tester. Following factors can be used to guess the errors:**

* Lessons learnt from past releases
* Historical learning
* Previous defects
* Production tickets
* Review checklist
* Application UI
* Previous test results
* Risk reports of the application
* Variety of data used for testing.

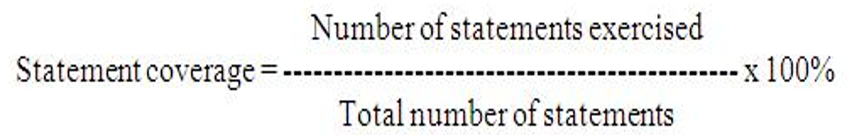
**White box testing techniques:** White box testing requires good knowledge of technology to do this testing.

Below are few of white box testing techniques:

**Structural Testing:**

**Statement Coverage Testing:**

* The statement coverage is also known as line coverage or segment coverage.
* The statement coverage **covers only the true conditions.**
* Through statement coverage we can identify the statements executed and where the code is not executed because of blockage.
* In this process each and every line of code needs to be checked and executed
* It verifies what the written code is expected to do and not to do
* It measures the quality of code written
* It checks the flow of different paths in the program and it also ensure that whether those path are tested or not.
* It cannot test the false conditions.
* It does not report that whether the loop reaches its termination condition.
* It does not understand the logical operators.



Consider code sample 4.1 :  
READ X  
READ Y  
I F X>Y

THEN Z = 50  
ENDIF  
**Code sample 4.1**

To achieve 100% statement coverage of this code segment just one test case is required, one which ensures that variable X contains a value that is greater than the value of variable Y, for example, X = 12 and Y = 10. Note that here we are doing structural test *design* first, since we are choosing our input values in order ensure statement coverage.

Now, let’s take another example where we will measure the coverage first. In order to simplify the example, we will regard each line as a statement. A statement may be on a single line, or it may be spread over several lines. One line may contain more than one statement, just one statement, or only part of a statement. Some statements can contain other statements inside them. In code sample 4.2, we have two read statements, one assignment statement, and then one IF statement on three lines, but the IF statement contains another statement (print) as part of it.

1 READ X  
2 READ Y  
3 Z =X + 2\*Y  
4 IF Z> 50 THEN  
5 PRINT large Z  
6 ENDIF

**Code sample 4.2**

Although it isn’t completely correct, we have numbered each line and will regard each line as a statement. Let’s analyze the coverage of a set of tests on our six-statement program:

TEST SET 1   
Test 1\_1: X= 2, Y = 3  
Test 1\_2: X =0, Y = 25  
Test 1\_3: X =47, Y = 1

Which statements have we covered?

* In Test 1\_1, the value of Z will be 8, so we will cover the statements on lines 1 to 4 and   line 6.
* In Test 1\_2, the value of Z will be 50, so we will cover exactly the same statements as Test 1\_1.
* In Test 1\_3, the value of Z will be 49, so again we will cover the same statements.

Since we have covered five out of six statements, we have 83% statement coverage (with three tests). What test would we need in order to cover statement 5, the one statement that we haven’t exercised yet? How about this one:

Test 1\_4: X = 20, Y = 25

This time the value of Z is 70, so we will print ‘Large Z’ and we will have exercised all six of the statements, so now statement coverage = 100%. Notice that we measured coverage first, and then designed a test to cover the statement that we had not yet covered.

Note that Test 1\_4 on its own is more effective which helps in achieving 100% statement coverage, than the first three tests together. Just taking Test 1\_4 on its own is also more efficient than the set of four tests, since it has used only one test instead of four. Being more effective and more efficient is the mark of a good test technique

Condition Coverage Testing:

Condition coverage is also known as Predicate Coverage in which each one of the Boolean expression have been evaluated to both TRUE and FALSE.

Example

if ((A || B) && C)

{

<< Few Statements >>

}

else

{

<< Few Statements >>

}

Result

In order to ensure complete Condition coverage criteria for the above example, A, B and C should be evaluated at least once against "true" and "false".

So, in our example, the 3 following tests would be sufficient for 100% Condition coverage testing.

A = true | B = false | C = false

A = false | B = true | C = true

A = false | B = false | C = false

**Branch Testing:**

Branch coverage is a testing method, which aims to ensure that each one of the possible branch from each decision point is executed at least once and thereby ensuring that all reachable code is executed.

* Branch coverage is also known as Decision coverage or all-edges coverage.
* It **covers both the true and false conditions** unlikely the statement coverage.
* A branch is the outcome of a decision, so **branch coverage** simply measures which decision outcomes have been tested. This sounds great because it takes a more in-depth view of the source code than simple statement coverage
* A decision is an IF statement, a loop control statement (e.g. DO-WHILE or REPEAT-UNTIL), or a CASE statement, where there are two or more outcomes from the statement. With an IF statement, the exit can either be TRUE or FALSE, depending on the value of the logical condition that comes after IF.

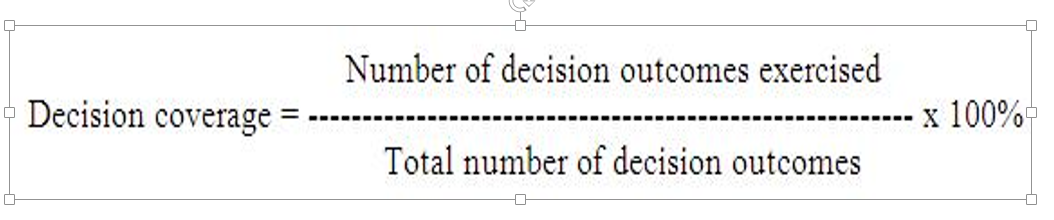
**Advantages of decision coverage:**

* To validate that all the branches in the code are reached
* To ensure that no branches lead to any abnormality of the program’s operation
* It eliminate problems that occur with statement coverage testing

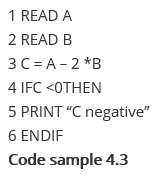
**Disadvantages of decision coverage:**

* This metric ignores branches within boolean expressions which occur due to short-circuit operators.

The decision coverage can be calculated as given below:



In the previous section we saw that just one test case was required to achieve 100% statement coverage. However, decision coverage requires each decision to have had both a True and False outcome. Therefore, to achieve 100% decision coverage, a second test case is necessary where A is less than or equal to B which ensures that the decision statement ‘IF A > B’ has a False outcome. So one test is sufficient for 100% statement coverage, but two tests are needed for 100% decision coverage. It is really very important to note that **100% decision coverage guarantees 100% statement coverage, but *not* the other way around.**

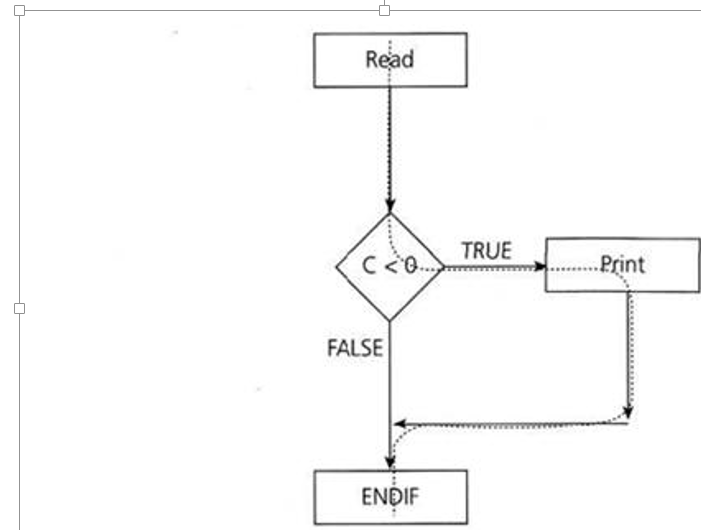


Let’s suppose that we already have the following test, which gives us 100% statement coverage for code sample 4.3.

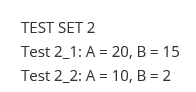
TEST SET 2   Test 2\_1: A = 20, B = 15

The value of C is -10, so the condition  ‘C < 0′ is True, so we will print ‘C negative’ and we have executed the True outcome from that decision statement. But we have not executed the False outcome of the decision statement. What other test would we need to exercise the False outcome and to achieve 100% decision coverage?

Before we answer that question, let’s have a look at another way to represent this code. Sometimes the decision structure is easier to see in a control flow diagram (see Figure 4.4).



The dotted line shows where Test 2\_1 has gone and clearly shows that we haven’t yet had a test that takes the False exit from the IF statement.  
Let’s modify our existing test set by adding another test:

  
This now covers both of the decision outcomes, True (with Test 2\_1) and False (with Test 2\_2). If we were to draw the path taken by Test 2\_2, it would be a straight line from the read statement down the False exit and through the ENDIF. We could also have chosen other numbers to achieve either the True or False outcomes.

Example:

Read A

Read B

IF A+B > 10 THEN

Print "A+B is Large"

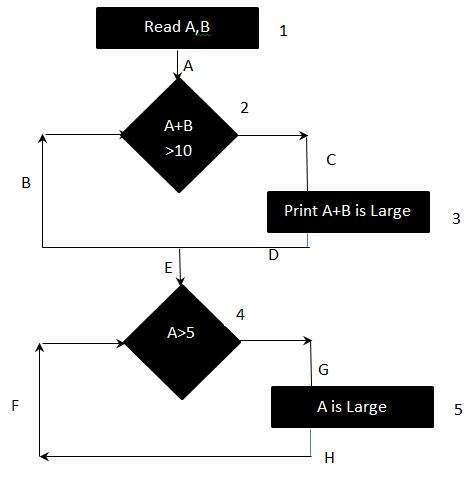
ENDIF

If A > 5 THEN

Print "A Large"

ENDIF

The above logic can be represented by a flowchart as:



Result :

To calculate Branch Coverage, one has to find out the minimum number of paths which will ensure that all the edges are covered. In this case there is no single path which will ensure coverage of all the edges at once. The aim is to cover all possible true/false decisions.

(1) 1A-2C-3D-E-4G-5H

(2) 1A-2B-E-4F

Hence Branch Coverage is 2.

(1) 1A-2C-3D-E-4G-5H

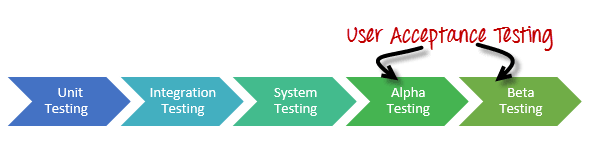
(2) 1A-2B-E-4F

Hence Branch Coverage is 2.

**Levels of Testing:**

We have below levels of testing in software testing:

* 1. Unit Testing
  2. Integration Testing
  3. System Testing
  4. UAT- User Accepting Testing



**Unit Testing:** Unit of code which is written by individual developer. This testing is performed by developer to make whatever code is written working fine and not having any defects

**Integration Testing:** Here we integrate / merge / combine every developer codeand do testing to see whether each component is interacting with other component with out any issues. Integration testing is performed by developers but they can ask from testing help so it can be performed by developer and testing team.

We have below approaches in integration testing

* Top Down
* Bottom Up

**Top Down Integration:**

Top down integration testing is an incremental integration testing technique which begins by testing the top-level module and progressively adds in lower level module one by one. Lower level modules are normally simulated by stubs, which mimic functionality of lower level modules. As you add lower level code, you will replace stubs with the actual components.

Top Down integration can be performed and tested in breadth first or depth firs manner.

**Advantages**:

• Drivers do not have to be written when top down testing is used. • It provides early working module of the program and so design defects can be Found and corrected early.

**Disadvantages**:

• Stubs have to be written with utmost care, as they will simulate setting of output parameters.

• It is difficult to have other people or third parties to perform this testing

Mostly Developers will have to spend some time on this

**Bottom Up Integration:**

In bottom up integration testing, module at the lowest level are developed first and other modules which go towards the 'main' program are integrated and tested one at a time. Bottom up integration also uses test drivers to drive and pass appropriate data to the lower level modules. As and when code for other module gets ready, these drivers are replaced with the actual module.

In this approach, lower level modules are tested extensively thus make sure that highest used module is tested properly.

Advantages:

Behaviours of the interaction points are crystal clear, as components are added in the controlled manner and tested repetitively. • Appropriate for applications where bottom up design methodology is used

Disadvantages:

Writing and maintaining test drivers or harness is difficult than writing stubs. • This approach is not suitable for the software development using top down approach **Top Down Integration:**

Top down integration testing is an incremental integration testing technique which begins by testing the top-level module and progressively adds in lower level module one by one. Lower level modules are normally simulated by stubs, which mimic functionality of lower level modules. As you add lower level code, you will replace stubs with the actual components.

Top Down integration can be performed and tested in breadth first or depth firs manner.

**Advantages**:

• Drivers do not have to be written when top down testing is used. • It provides early working module of the program and so design defects can be Found and corrected early.

**Disadvantages**:

• Stubs have to be written with utmost care, as they will simulate setting of output parameters.

• It is difficult to have other people or third parties to perform this testing

Mostly Developers will have to spend some time on this

**Bottom Up Integration:**

In bottom up integration testing, module at the lowest level are developed first and other modules which go towards the 'main' program are integrated and tested one at a time. Bottom up integration also uses test drivers to drive and pass appropriate data to the lower level modules. As and when code for other module gets ready, these drivers are replaced with the actual module.

In this approach, lower level modules are tested extensively thus make sure that highest used module is tested properly.

Advantages:

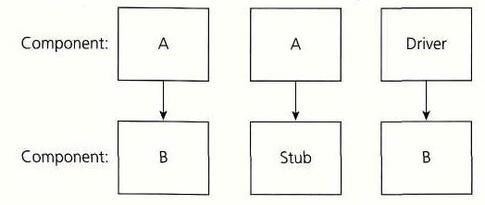
Behaviours of the interaction points are crystal clear, as components are added in the controlled manner and tested repetitively. • Appropriate for applications where bottom up design methodology is used

Disadvantages:

Writing and maintaining test drivers or harness is difficult than writing stubs. • This approach is not suitable for the software development using top down approach

In Top Down approach we use stubs and in bottom-up we use drivers. Let’s try to understand stub and drivers.

**What is the difference between stubs and drivers in software testing?**



The concept of Stubs and Drivers are mostly used in the case of component testing. Component testing may be done in isolation with the rest of the system depending upon the context of the development cycle.  
  
Stubs and drivers are used to replace the missing software and simulate the interface between the software components in a simple manner.  
  
Suppose you have a function (Function A) that calculates the total marks obtained by a student in a particular academic year. Suppose this function derives its values from another function (Function b) which calculates the marks obtained in a particular subject.  
  
You have finished working on Function A and wants to test it. But the problem you face here is that you can't seem to run the Function A without input from Function B; Function B is still under development. In this case, you create a dummy function to act in place of Function B to test your function. This dummy function gets called by another function. Such a dummy is called a Stub.  
  
To understand what a driver is, suppose you have finished Function B and is waiting for Function A to be developed. In this case you create a dummy to call the Function B. This dummy is called the driver.

**System Testing:** Once integration testing is completed we do system testing. Here we consider entire system as whole and do testing. Testing team is responsible to do system testing. In this testing we consider below types of testing’s. We will discuss these testing types in detail

* Smoke Testing / Build Verification Testing
* Exploratory Testing
* Compatibility Testing
* Installation Testing
* Un-Installation Testing
* Regression Testing
* Ad-hoc Testing
* Sanity Testing
* Usability Testing
* Alpha Testing

**UAT**: Is known as User Acceptance Testing and this testing is performed by client. We will have two types of testing in this:

**Alpha Testing:** This testing is performed by client in UAT environment and client is responsible for this testing.

**Beta Testing:** This testing is performed by end user in real time environment. We will discuss in detail about this in our notes.

**Popular types of testing types under system testing:**

**Smoke Test / Build verification test**: This testing is performed before going to complete system testing to make sure system is really ready to do further testing.

**Compatibility Testing:** This testing is performed to make sure system is compatible with below

1. Different operating systems:
   1. What are the operating systems on which we need to test the application
   2. What are the devices on which we have to test
   3. What are the versions of operating system which we have to use for testing
2. Different browsers:
   1. What are the browsers on which we need to test
   2. What are the versions of the browsers which are used for testing

**Regression Testing:**

This testing we do in below cases to make sure existing is not impacted due to below.

1. New code is added on top of existing code
2. Code is getting modified because of enchantments

Let’s take an example below

Sprint 1: Let’s say I have written 200 test cases which takes 1 month

1. System should allow user to login with valid user id and password
2. System should display an error message if user enters wrong password
3. System should lock account if user enters wrong password more than 3 times for 24 hours
4. System should allow adding beneficiary and should be with in the same bank
5. System should allow to transfer funds within the bank only

Sprint 2: Let’s say you have written 150 test cases which takes 1 month

1. User should be allowed to raise cheque book request through online
2. User should be allowed to transfer funds to other bank as well
3. User should be able to pay bills online
4. User should be able pay credit card bills as well
5. Smoke & Sanity Testing:

| **Smoke Testing** | **Sanity Testing** |
| --- | --- |
| Smoke Testing is performed to ascertain that the critical functionalities of the program is working fine | Sanity Testing is done to check the new functionality / bugs have been fixed |
| The objective of this testing is to verify the "stability" of the system in order to proceed with more rigorous testing | The objective of the testing is to verify the "rationality" of the system in order to proceed with more rigorous testing |
| This testing is performed by the developers or testers | Sanity testing is usually performed by testers |
| Smoke testing is usually not documented or scripted | Sanity testing is usually documented and is unscripted |
| Smoke testing is a subset of Regression testing | Sanity testing is a subset of Acceptance testing |
| Smoke testing exercises the entire system from end to end | Sanity testing exercises only the particular component of the entire system |
| Smoke testing is like General Health Check Up | Sanity Testing is like specialized health check up |

**Exploratory Testing:** This testing is performed by testing team. We do this testing while exploring the application. Example let’s say you have net banking application we do all online transactions while exploring the application. Let’s say you want to transfer funds we only explore how to do funds transfer and learn while doing the testing.

**Installation Testing:** This type testing we do only for applications / products which needs installations. We do this testing to make sure when end user tries to do installation of product it works fine without any issues. Example let’s say you are working for client Microsoft and doing testing of MS-Office, so you need to di installation testing to make sure this product goes with installation without any defects. Another example we do lot of app’s installation on our smart phone in our life but before it is coming to us there will be a dedicated testing team who do installation of these app’s before coming to real time world usage.

**Un-Installation Testing:** This testing we do after installation testing to make sure whatever software is installed can uninstalled as well without having any issues.

**Alpha Testing:**

Alpha testing is a type of acceptance testing; performed to identify all possible issues/bugs before releasing the product to everyday users or public.  The focus of this testing is to simulate real users by using blackbox and whitebox techniques. The aim is to carry out the tasks that a typical user might perform. Alpha testing is carried out in a lab(UAT) environment and usually the testers are internal employees of the client. To put it as simple as possible, this kind of testing is called alpha only because it is done early on, near the end of the development of the software, and before beta testing.

**Beta Testing:**

Beta Testing of a product is performed by "real users" of the software application in a "real environment" and can be considered as a form of external user acceptance testing.

 Beta version of the software is released to a limited number of end-users of the product to obtain feedback on the product quality. Beta testing reduces product failure risks and provides increased quality of the product through customer validation.

It is the final test before shipping a product to the customers. Direct feedback from customers is a major advantage of Beta Testing. This testing helps to tests the product in real time environment.

## Alpha Testing versus Beta testing:

Following are the differences of Alpha and Beta Testing

| **Alpha Testing** | **Beta Testing** |
| --- | --- |
| Alpha testing performed by Testers  who are usually internal employees of the client | Beta testing is performed by End Users who are not employees of the client |
| Alpha Testing performed at client site | Beta testing is performed at client location or end user of the product |
| Reliability and security testing are not performed  in-depth Alpha Testing | Reliability, Security, Robustness are checked during Beta Testing |
| Alpha testing involves both the white box and black box techniques | Beta Testing typically uses black box testing |
| Alpha testing requires lab environment(UAT) or testing environment | Beta testing doesn't require any lab environment or testing environment. Software is made available to the public and  is said to be real time environment |
| Long execution cycle may be required for Alpha testing | Only few weeks of execution are required for Beta testing |
| Critical issues or fixes can be addressed by developers immediately in Alpha testing | Most of the issues or feedback is collected from Beta testing will be implemented in future versions of the product |
| Alpha testing is to ensure the quality of the product before moving to Beta testing | Beta testing also concentrates on quality of the product, but gathers end users input on the product and ensures that the product is ready for real time users. |

**Non-Functional Testing:** As name says here we are not doing any functional testing and we are doing non-functional testing. It means we do performance testing where we see application response time like how much time it is taking to respond to user. We have below types of testing under performance testing

**Load Testing:**

Load testing is meant to test the system by constantly and steadily increasing the load on the system till the time it reaches the threshold limit. It is the simplest form of testing which employs the use of automation tools such as LoadRunner or any other good tools, which are available. Load testing is also famous by the names like volume testing and endurance testing.

Example:

For example, to check the email functionality of an application, it could be flooded with 1000 users at a time. Now, 1000 users can fire the email transactions (read, send, delete, forward, reply) in many different ways. If we take one transaction per user per hour, then it would be 1000 transactions per hour. By simulating 10 transactions/user, we could load test the email server by occupying it with 10000 transactions/hour.

**Stress Testing**: This is also a type of performance testing in which the application is operated above the threshold of break to check at what point that application crashes. Similar to load testing, server peak performance throughputs and response times are monitored.

Stress testing is primarily used during recovery testing, where in the application traffic is steadily increased above the threshold of break to monitor the application crash and how well the application recovers from the crash.

**Volume testing** checks if the system behaves as expected for certain volume of data. Volume can be increasing size of the file. On the other hand, load tests checks the performance of the system when the load is increased. Load testing here can be increasing the number of files. Volume testing can be used to measure the throughput while load testing can be used for measuring performance.

**Test Cases:**

Test Case: which consist of steps and expected result for every test step. It helps testing team to follow these steps to test specific functionality.

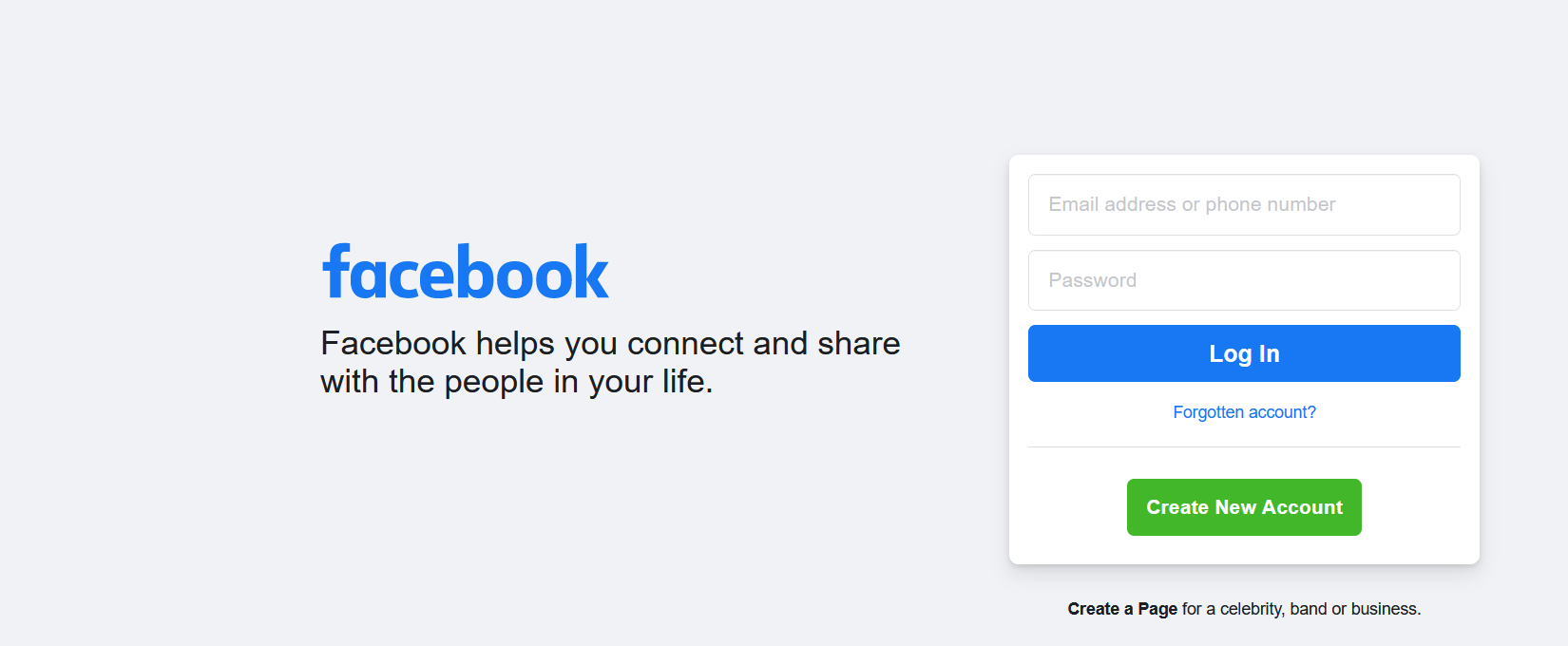
Test contains below:

1. Test Case Name
2. Pre-condition
3. Sr No
4. Test Step
5. Test Data
6. Expected Results
7. Actual Result
8. Status
9. Comments
10. Defect Id
11. Executed By
12. Execution Date

Types of test cases:

1. GUI(Graphical User Interface): Here we write test cases for below
   1. Check UI is appearing as the requirements given by the client
   2. Check any alignment issues
   3. Check for spelling mistakes
   4. Overlapping issues
2. Functional Test Cases: Here we write test cases to check the functionalities of application like check user is able to register an account, check user is able reset password etc…
3. Field Level Validation: Here we write test cases to check for each field for below validations
   1. If a filed is a mandatory leave that field empty click submit button
   2. if field should take minimum of 5 letters then enter less than 5 letters
   3. if field should take maximum of 100 letters then enter more than 100 letters
   4. if field should take only letters then enter some special symbols like +\_)(\*&^^%%

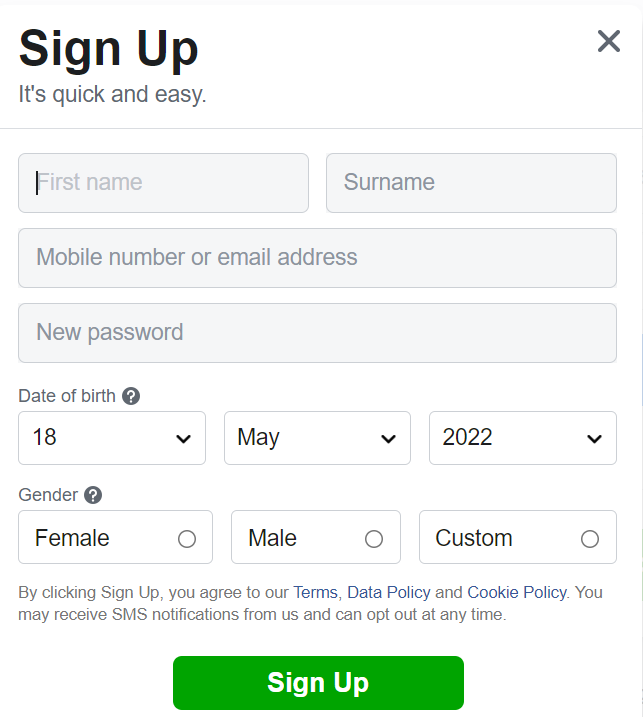
Facebook home page should be displayed as per below:



Let’s say email address or phone number requirements

1. This field mandatory
2. If user entering numeric data for phone number it should be max of 12 digits
3. If user enters less than 12 digits system should display an error message stating "phone number must be 12 digits"
4. If email address or phone number is not entered system should display an error message stating “"email address or phone number is required field"

Create New Account screen should as per the below:



Field level requirements:

* 1. First Name:
     + This field is mandatory and if user does not enter data it should display error message stating: “This filed is required”
     + This filed should take maximum of 100 letters
     + This should accept only letters if user enters other than letters it should display an error message stating “First Name field should be minimum of 5 letters and maximum of 100 letters only”
     + T This should be minimum of 5 letters if user enters less than 5 letters it should display an error message stating “First Name field should be minimum of 5 letters and maximum of 100 letters only”
     + If user enters other than letters system should display an error message stating “First Name accept only letters”

**Test Environments:**

All most all the projects will have below environments:

1. Test Environment: This environment is used by testing team to do system testing.
2. UAT Environment: This environment is used by client team to do alpha testing.
3. Pre-Production: This environment is a replica of production and it is used for doing the performance testing
4. Production: This environment real time environment and used by end users. This environment also used for doing beta testing.

Let’s take an example that you are working for FaceBook application then below URL’s will be used for each environment

Test Environment: Facebooktest.com

UAT Environment: facebookuat.com

Pre-Production: facebookpreprod.com

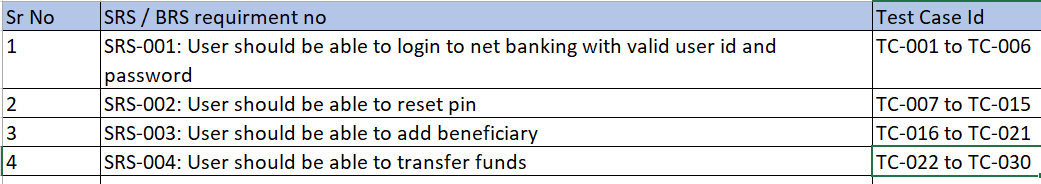
Production: facebook.com

#### Traceability Matrix

Requirement Traceability Matrix or RTM captures all requirements proposed by the client or development team and their traceability in a single document delivered at the conclusion of the life-cycle.

In other words, it is a document that maps and traces user requirement with test cases. The main purpose of Requirement Traceability Matrix is to see that all test cases are covered so that no functionality should miss while testing.

Template of the traceability matrix looks as below:



**STLC (Software Testing Life Cycle):**

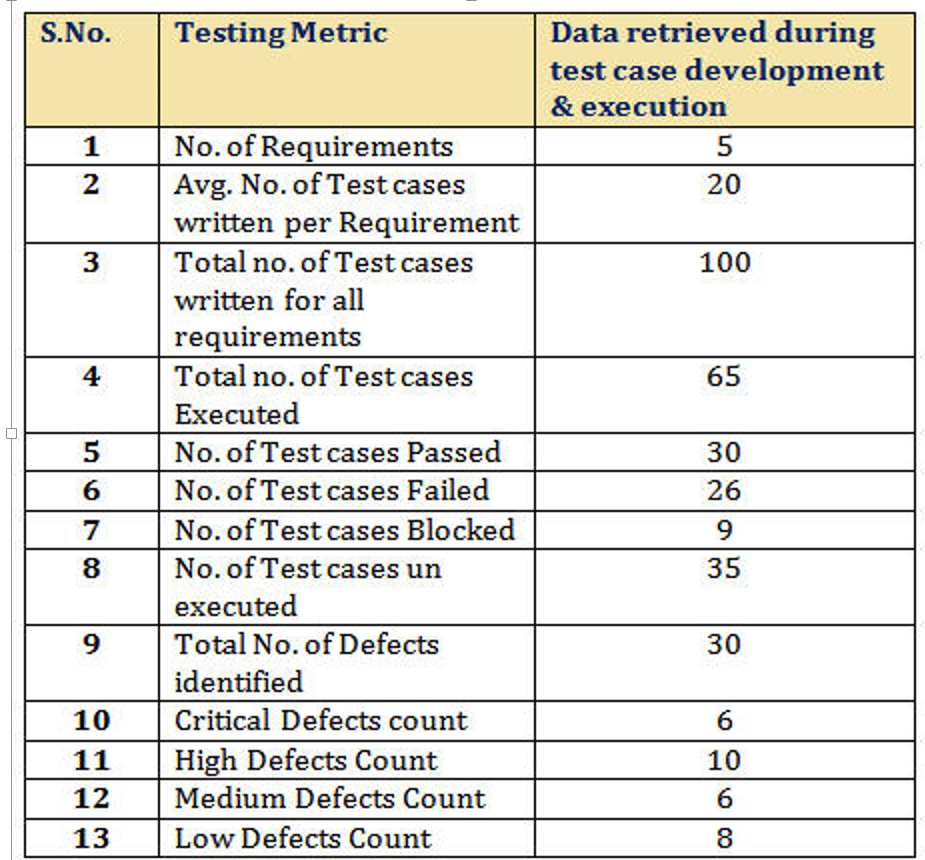
1. Understand SRS / BRS / Requirements / User Stories
2. Raise questions and get it clarified
3. Prepare Test Plan
4. Prepare Test Cases
5. Review the Test Cases
6. Automation
7. Execute Test Cases
8. Defect logging
9. Defect Retest
10. Execute automation test scripts as part of regression testing

**Test Metrics:**

In software testing, **Metric** is a quantitative measure of the degree to which a **system, system component, or process** possesses a given attribute.

In other words, metrics helps estimating the progress, quality and health of a software testing effort. The ideal example to understand metrics would be a weekly mileage of a car compared to its ideal mileage recommended by the manufacturer.

**Below is the table format for the data retrieved from the test analyst who is actually involved in testing:**



**Definitions and Formulas for Calculating Metrics:**

**#1) %ge Test cases Executed**: This metric is used to obtain the execution status of the test cases in terms of %ge.

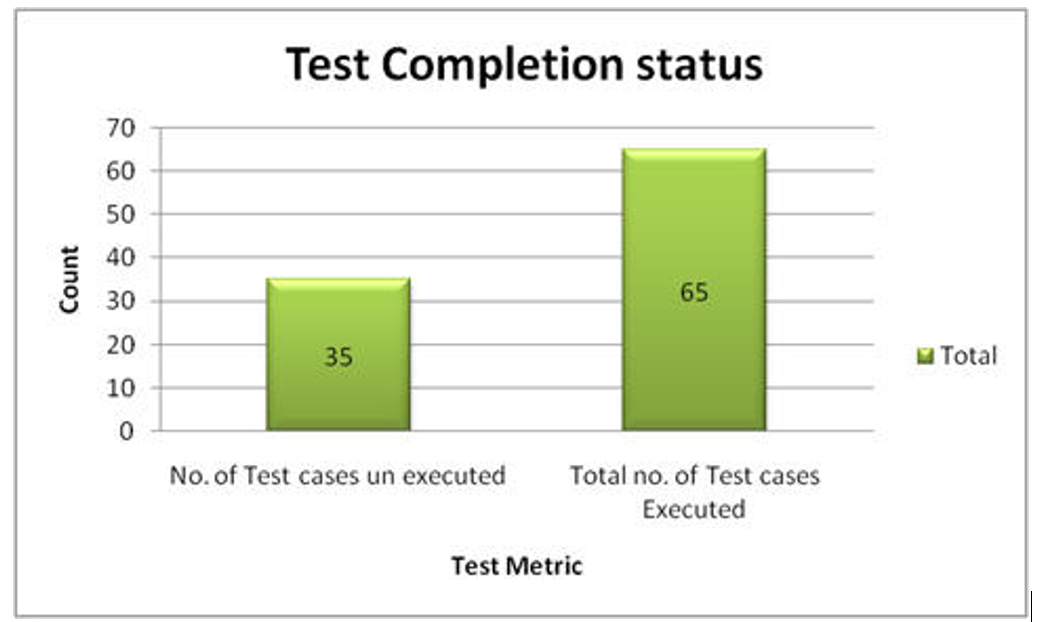
%ge Test cases Executed = **(**No. of Test cases executed / Total no. of Test cases written) \* 100.

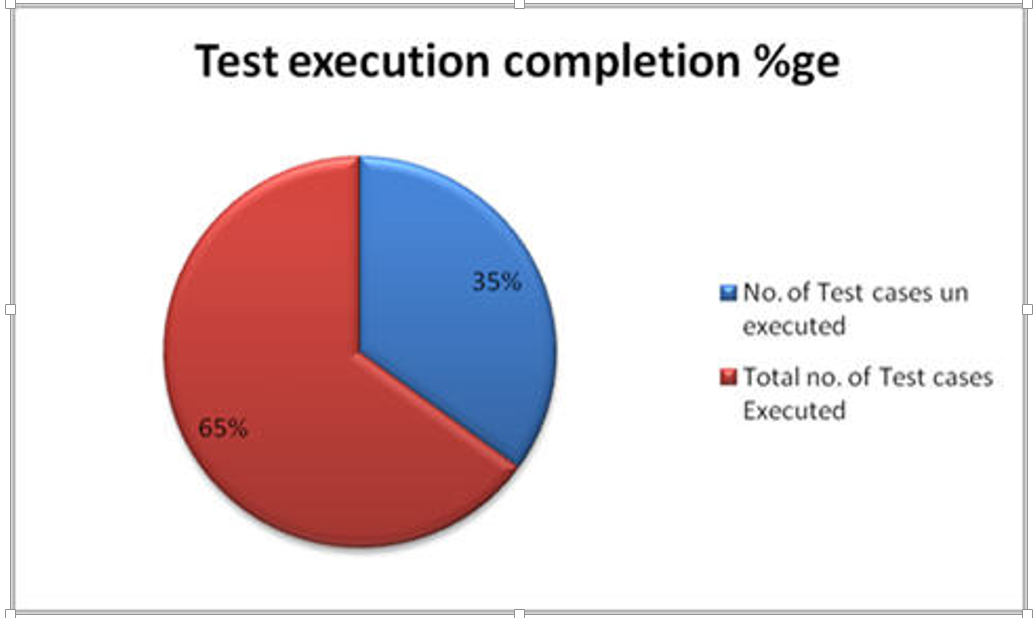
So, from the above data,  
%ge Test cases Executed = (65 / 100) \* 100 = 65%

**#2) %ge Test cases not executed**: This metric is used to obtain the pending execution status of the test cases in terms of %ge.

%ge Test cases not executed = **(**No. of Test cases not executed / Total no. of Test cases written) \* 100.

So, from the ct executed = (35 / 100) \* 100 = 35%





**#3) %ge Test cases Passed**: This metric is used to obtain the Pass %ge of the executed test cases.

%ge Test cases Passed = **(**No. of Test cases Passed / Total no. of Test cases Executed) \* 100.

So, from the above data,  
%ge Test cases Passed = (30 / 65) \* 100 = 46%

**#4) %ge Test cases Failed**: This metric is used to obtain the Fail %ge of the executed test cases.

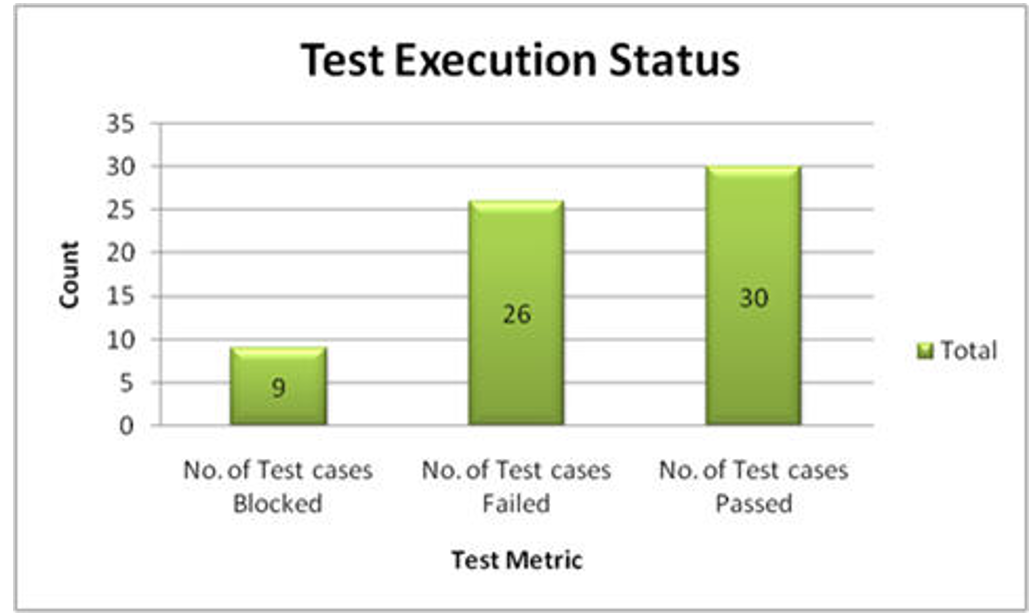
%ge Test cases Failed = **(**No. of Test cases Failed / Total no. of Test cases Executed) \* 100.

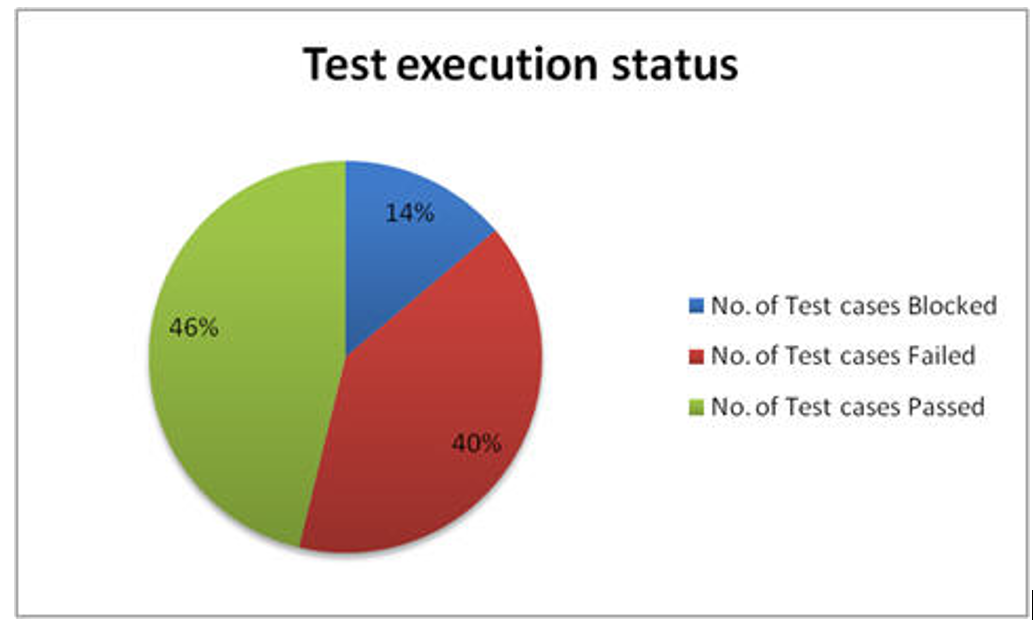
So, from the above data,  
%ge Test cases Passed = (26 / 65) \* 100 = 40%

**#5) %ge Test cases Blocked**: This metric is used to obtain the blocked %ge of the executed test cases. A detailed report can be submitted by specifying the actual reason of blocking the test cases.

%ge Test cases Blocked = **(**No. of Test cases Blocked / Total no. of Test cases Executed) \* 100.

So, from the above data,  
%ge Test cases Blocked = (9 / 65) \* 100 = 14%





**#6) Defect Density =** No. of Defects identified / size

**(**Here “Size” is considered as requirement. Hence here the Defect Density is calculated as number of defects identified per requirement. Similarly, Defect Density can be calculated as number of Defects identified per 100 lines of code [OR] No. of defects identified per module etc.**)**

So, from the above data,  
Defect Density = (30 / 5) = 6

**#7) Defect Removal Efficiency (DRE) = (**No. of Defects found during QA testing / (No. of Defects found during QA testing +No. of Defects found by End user)) \* 100

DRE is used to identify the test effectiveness of the system.  
Suppose, During Development & QA testing, we have identified 100 defects.  
After the QA testing, during Alpha & Beta testing, end user / client identified 40 defects, which could have been identified during QA testing phase.

Now, The DRE will be calculated as,  
DRE = [100 / (100 + 40)] \* 100 = [100 /140] \* 100 = 71%

**$8) Defect Leakage:** Defect Leakage is the Metric which is used to identify the efficiency of the QA testing i.e., how many defects are missed / slipped during the QA testing.

Defect Leakage **= (**No. of Defects found in UAT / No. of Defects found in QA testing.) \* 100

Suppose, During Development & QA testing, we have identified 100 defects.  
After the QA testing, during Alpha & Beta testing, end user / client identified 40 defects, which could have been identified during QA testing phase.

Defect Leakage = (40 /100) \* 100 = 40%

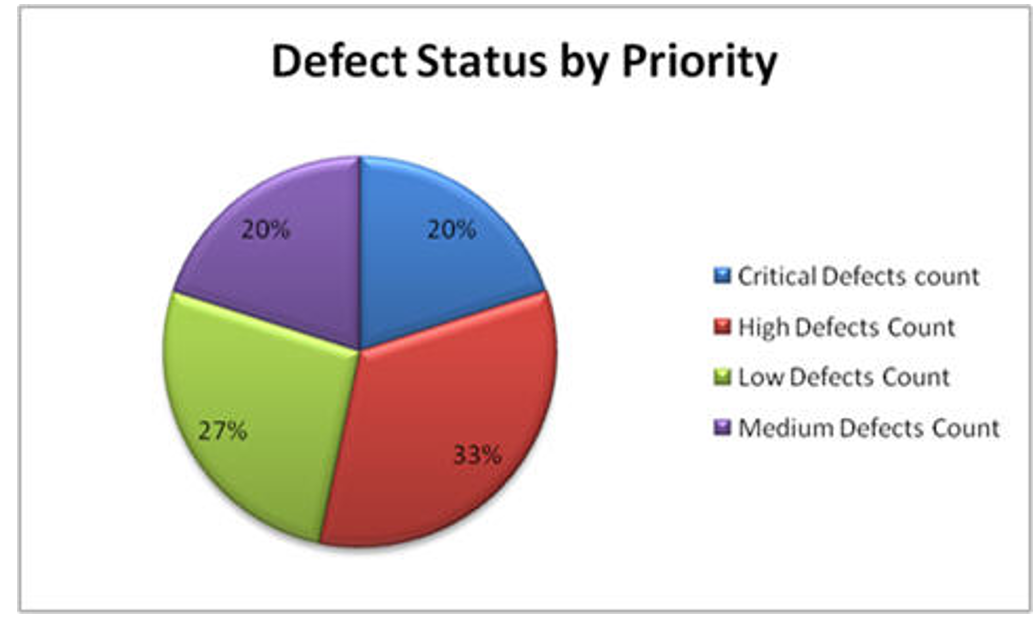
**#9) Defects by Priority**: This metric is used to identify the no. of defects identified based on the Severity / Priority of the defect which is used to decide the quality of the software.

%ge Critical Defects = No. of Critical Defects identified / Total no. of Defects identified \* 100  
From the data available in the above table,  
%ge Critical Defects = 6/ 30 \* 100 = 20%

%ge High Defects = No. of High Defects identified / Total no. of Defects identified \* 100  
From the data available in the above table,  
%ge High Defects = 10/ 30 \* 100 = 33.33%

%ge Medium Defects = No. of Medium Defects identified / Total no. of Defects identified \* 100  
From the data available in the above table,  
%ge Medium Defects = 6/ 30 \* 100 = 20%

%ge Low Defects = No. of Low Defects identified / Total no. of Defects identified \* 100  
From the data available in the above table,  
%ge Low Defects = 8/ 30 \* 100 = 27%



**Defect Management:**

**Defect , Error , Bug and Failure?**

A mistake in coding is called error,error found by tester is called defect, defect accepted by development team then it is called bug ,build does not meet the requirements then it Is failure.

**What needs to be considered before creating defect:**

* 1. Check if any duplicate defect exist (refer defect dash board for this or search in the title)
  2. Put valid title by which it can be understandable
  3. Provide clear details in description as much as possible
  4. Include requirement reference number in each defect by which it makes easier to activate
  5. Attach screen shots and video if applicable to understand easily
  6. Provide detail steps along with test data used to reproduce the defect
  7. Make sure to use correct “Iteration Path” and “Area Path”
  8. Use correct “Build Number” in which it is found
  9. Use correct severity and priority of the defect as defined (consider workaround before doing this)

**Defining severity of the defect**

Severity of the defect is decided based on how severely it is impacting to system

Below are the defect severities:

Critical: Show stopper, blocking to proceed, having huge financial impact etc

High: Having direct impact on the customer or one of the core functionality is not working

Medium: Not having impact on functional side or not impacting to customer like slight difference in displaying the error message

Low: Like color is not displaying as expected or capital letter is missing the label or spelling mistake

**Defining priority of the defect:**

Priority is decided based on urgency of resolution for a defect

Below are the defect priorities:

P1: Fix is required immediately

P2: Not immediately but required in next planned release

P3: Not planned for any release but will be fixed in future releases

**Defect Life Cycle:**

Proposed / New: This is the status when defect is newly created by testing team

Triage: Testing Lead / Defect Lead reviewed defect and ready for discussions with stake holders.

Active: Development Lead / Project Lead agrees for activation of the defect

In progress: Individual developer started working on it

Resolved: Individual developer fixed and assigned to tester to re-test

Re-Open: Tester re-opnes the defect as it still not working even after fix

Rejected: Rejected by development team stating not a valid bug (like duplicate bug, not reproducible, working as expected)

Closed: Tested by test team and closed (working as expected) or agreed with rejection reason