Genba Sopanrao Moze Trust's
Parvatibai Genba Moze College of Engineering

Department of MCA

LABORATORY MANUAL

For the Academic Year 2024 - 2025

SYMCA- Semester III

Teaching Scheme:

PR: 04Hours/Week **Scheme: TW:25Marks**

PR: 50 Marks

PROGRAM OUTCOMES	
PO NO	**Program Outcome Description**
Po 1	Apply knowledge of mathematics, computer science, computing specializations appropriate for real world applications.
Po 2	Identify, formulate, analyze and solve complex computing problems using relevant domain disciplines.
Po 3	Design and evaluate solutions for complex computing problems that meet specified needs with appropriate considerations for real world problems.
Po 4	Find solutions of complex computing problems using design of experiments, analysis and interpretation of data.
Po 5	Apply appropriate techniques and modern computing tools for development of complex computing activities.
Po 6	Apply professional ethics, cyber regulations and norms of professional computing practices.
Po 7	Recognize the need to have ability to engage in independent and life-long learning in the broadest context of technological change.
Po 8	Demonstrate knowledge and understanding of the computing and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
Po 9	Demonstrate knowledge and understanding of the computing and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
Po 10	Assess societal, environmental, health, safety, legal and cultural issues within local and global contexts, and the consequent responsibilities relevant to the professional computing practices
Po 11	Assess societal, environmental, health, safety, legal and cultural issues within local and global contexts, and the consequent responsibilities relevant to the professional computing practices
Po 12	Identify a timely opportunity and use innovation, to pursue opportunity, as a successful Entrepreneur /professional

OBJECTIVE:

- To understand the fundamental concepts of database management. These concepts include aspects
- of database design, database languages, and database-system implementation.
- To provide a strong formal foundation in database concepts, technology and practice.
- To give systematic database design approaches covering conceptual design, logical design and an
- overview of physical design.
- Be familiar with the basic issues of transaction processing and concurrency control.
- To learn and understand various Database Architectures and Applications.
- To learn a powerful, flexible and scalable general purpose database to handle big data.

Course Outcomes	
CO 1	Design E-R Model for given requirements and convert the same into database tables
CO 2	Use database techniques such as SQL & PL/SQL
CO 3	Use modern database techniques such as NOSQL.
CO 4	Explain transaction Management in relational database System.
CO 5	Describe different database architecture and analyses the use of appropriate architecture in real time environment.
CO 6	Students will be able to use advanced database Programming concepts Big Data – HADOOP
CO 7	Interpret the importance of Computational Intelligence for solving the different problems

Guidelines for Student Journal:

The laboratory assignments are to be submitted by student in the form of journal. Journal consists of prologue, Certificate, table of contents, and **handwritten write-up** of each assignment (Title, Objectives, Problem Statement, Outcomes, software & Hardware requirements, Date of Completion, Assessment grade/marks and assessor's sign, Theory- Concept in brief, algorithm, flowchart, Design, test cases, conclusion/analysis.

Program codes with sample output of all performed assignments are to be submitted as softcopy.

As a conscious effort and little contribution towards Green IT and environment awareness, attaching printed papers as part of write-ups and program listing to journal may be avoided. Use of DVD containing students programs maintained by lab In-charge is highly encouraged. For reference one or two journals may be maintained with program prints at Laboratory.

EXPERIMENT NO. 01

ASSIGNMENT NO: 1

TITLE: Implementation of DDL commands of SQL with suitable examples

- Create table
- Alter table
- Drop Table

AIM: To study and execute the DDL commands in RDBMS

DDL aims to establish and modify the structure of objects stored in a database.

These definitions and modifications control descriptions of the database schema.

Outcomes:

SQL DDL commands are used for creating new database objects (CREATE command), modifying existing database objects (ALTER command), and deleting or removing database objects (DROP and TRUNCATE commands).

Hardware and software Requirement:

Database: Oracle Database Server 12.1.0.2, Mandatory database components:

Oracle Text.

Oracle Java Virtual Machine (JVM).

Memory: 8 GB RAM or more

Query:

```
Create table student(roll number(3) primary key, name varchar(20)
not null, dateofbirth date, fees number(7,2));
```

Alter table

```
Alter table student add (address varchar2(50));
```

Drop Table

```
drop table student;
```

ASSIGNMENT No: 2

TITLE: Design test cases for any E-Commerce Website

- **AIM:** Design test cases for any E-Commerce Website

- **THEORY: Test Case:** A test case is a document, which has a set of test data, preconditions, expected results and post conditions, developed for a particular test scenario in order to verify compliance against a specific requirement. Test Case acts as the starting point for the test execution, and after applying a set of input values the application has a definitive outcome and leaves the system at some end point or also known as execution post condition.

Typical Test Case Parameters:

Test Suite ID	The ID of the test suite to which this test case belongs.
Test Case ID	The ID of the test case.
Test Case Summary	The summary / objective of the test case.
Related Requirement	The ID of the requirement this test case relates/traces to.
Prerequisites	Any prerequisites or preconditions that must be fulfilled prior to executing the test.
Test Procedure	Step-by-step procedure to execute the test.
Test Data	The test data, or links to the test data, that are to be used while conducting the test.
Expected Result	The expected result of the test.
Actual Result	The actual result of the test; to be filled after executing the test.
Status	Pass or Fail. Other statuses can be 'Not Executed' if testing is not performed and 'Blocked' if testing is blocked.

Remarks	Any comments on the test case or test execution.
Created By	The name of the author of the test case.
Date of Creation	The date of creation of the test case.
Executed By	The name of the person who executed the test.
Date of Execution	The date of execution of the test.
Test Environment	The environment (Hardware/Software/Network) in which the test was executed.

Test case for Login form:

Project Name:	
Test Case Template	
Test Case ID: LMSL	Test Designed by:
Test Priority (Low/Medium/High): Med	Test Designed date:
Module Name: LMSL login screen	Test Executed by:
Test Title: Verify Login with valid user name and Password.	Test Execution date:
Description: Test the LMSL Login Page	
Pre-conditions: User has valid username and password	
Dependencies:	

Step	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes
1	Enter Valid Username and Password	Username= Password=	User should able to login	Login successful and go to dash board	Pass	

2	Enter Valid username & blank Password	Username= Password=	Invalid Password	Invalid Password stay remains to login screen	Pass	
3	Enter invalid username & valid Password	Username= Password=	Invalid Username	Invalid username stay remains to login screen	Pass	
4	Enter invalid username & invalid password	Username= And password=	Invalid username & Password	Invalid Username and Password	Pass	

ASSIGNMENT No: 3A

TITLE: Write black box test cases for an application

- **AIM:** Write black box test cases for an application

- **THEORY:** Black Box Testing : (Types and techniques of BBT), 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. The term 'behavioral testing' is also used for black box testing and white box testing is also sometimes called 'structural testing'. Behavioral test design is slightly different from black-box test design because the use of internal knowledge isn't strictly forbidden, but it's still discouraged. Each testing method has its own advantages and disadvantages. There are some bugs that cannot be found using only black box or only white box. Majority of the application are tested by black box testing method. We need to cover majority of test cases so that most of the bugs will get discovered by black box testing. Black box testing occurs throughout the software development and Testing life cycle i.e. in Unit, Integration, System, Acceptance and regression testing stages.

Tools used for Black Box testing:

Black box testing tools are mainly record and playback tools. These tools are used for regression testing that to check whether new build has created any bug in previous working application functionality. These record and playback tools records test cases in the form of some scripts like TSL, VB script, Java script, Perl.

Advantages of Black Box Testing:

- Tester can be non-technical.
- Used to verify contradictions in actual system and the specifications.
- Test cases can be designed as soon as the functional specifications are complete.

Disadvantages of Black Box Testing:

- The test inputs needs to be from large sample space.
- It is difficult to identify all possible inputs in limited testing time. So writing test cases is slow and difficult.
- Chances of having unidentified paths during this testing.

Methods of Black box Testing:

- Graph Based Testing Methods: Each and every application is build up of some objects. All such objects are identified and graph is prepared. From this object graph each object relationship is identified and test cases written accordingly to discover the errors.
- Error Guessing: This is purely based on previous experience and judgment of tester. Error Guessing is the art of guessing where errors can be hidden. For this technique there are no specific tools, writing the test cases that cover all the application paths.
- Boundary Value Analysis: Many systems have tendency to fail on boundary. So testing boundary values of application is important. Boundary Value Analysis (BVA) is a test Functional Testing

technique where the extreme boundary values are chosen. Boundary values include maximum, minimum, just inside/outside boundaries, typical values, and error values. Extends equivalence partitioning. Test both sides of each boundary Look at output boundaries for test cases too Test min, min-1, max, max+1, typical values

BVA techniques:

1. Number of variables For n variables: BVA yields 4n + 1 test cases.
2. Kinds of ranges Generalizing ranges depends on the nature or type of variables Advantages of Boundary Value Analysis

1. Robustness Testing – Boundary Value Analysis plus values that go beyond the limits
2. Min – 1, Min, Min +1, Nom, Max -1, Max, Max +1
3. Forces attention to exception handling

Limitations of Boundary Value Analysis

Boundary value testing is efficient only for variables of fixed values i.e boundary.

- **Equivalence Partitioning:**

Equivalence partitioning is a black box testing method that divides the input domain of a program into classes of data from which test cases can be derived.

- **How is this partitioning performed while testing:**

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

- **Comparison Testing:**

Different independent versions of same software are used to compare to each other for testing in this method.

- **What is Black Box Testing?**

Firstly we will learn what is Black Box Testing? Here we will discuss about how black box testing is perform, different BBT Techniques used in testing.

- **Black Box Testing Method:**

Black box testing is the Software testing method which is used to test the software without knowing the internal structure of code or program. Most likely this testing method is what most of tester actual perform and used the majority in the practical life.

Basically software under test is called as “Black-Box”, we are treating this as black box & without checking internal structure of software we test the software. All testing is done as customer’s point of view and tester is only aware of what is software is suppose to do but how these requests are processing by

software is not aware. While testing tester is knows about the input and expected output's of the software and they do not aware of how the software or application actually processing the input requests & giving the outputs. Tester only passes valid as well as invalid inputs & determines the correct expected outputs. All the test cases to test using such method are calculated based on requirements & specifications document.

The main purpose of the Black Box is to check whether the software is working as per expected in requirement document & whether it is meeting the user expectations or not. There are different types of testing used in industry. Each testing type is having its own advantages & disadvantages. So fewer bugs cannot be find using the black box testing or white box testing.

- **Types of Black Box Testing Techniques:** Following black box testing techniques are used for testing the software application.
- Boundary Value Analysis (BVA)
- Equivalence Class Partitioning
- Decision Table based testing
- Cause-Effect Graphing Technique
- Error Guessing

1) Boundary Value Analysis (BVA):

Boundary Value Analysis is the most commonly used test case design method for black box testing. As all we know the most of errors occurs at boundary of the input values. This is one of the techniques used to find the error in the boundaries of input values rather than the center of the input value range. Boundary Value Analysis is the next step of the Equivalence class in which all test cases are design at the boundary of the Equivalence class. Let us take an example to explain this:

Suppose we have software application which accepts the input value text box ranging from 1 to 1000, in this case we have invalid and valid inputs:

Invalid Input	Valid Input	Invalid Input
0 – less	1 – 1000	1001 – above

Here are the Test cases for input box accepting numbers using Boundary value analysis:

Min value – 1	0
Min Value	1
Min value + 1	2
Normal Value	1 – 1000

Max value – 1	999
Max value	1000
Max value +1	1001

This is testing techniques is not applicable only if input value range is not fixed i.e. the boundary of input is not fixed.

2) Equivalence Class Partitioning

The equivalence class partition is the black box test case design technique used for writing test cases. This approach is use to reduce huge set of possible inputs to small but equally effective inputs. This is done by dividing inputs into the classes and gets one value from each class. Such method is used when exhaustive testing is most wanted & to avoid the redundancy of inputs. In the equivalence partitioning input are divided based on the input values:

- If input value is Range, then we one valid equivalence class & two invalid equivalence classes.
- If input value is specific set, then we one valid equivalence class & one invalid equivalence classes.
- If input value is number, then we one valid equivalence class & two invalid equivalence classes.
- If input value is Boolean, then we one valid equivalence class & one invalid equivalence classes.

Black Box Testing Example

In this technique, we do not use the code to determine a test suite; rather, knowing the problem that we're trying to solve, we come up with four types of test data:

1. Easy-to-compute data
2. Typical data
3. Boundary / extreme data
4. Bogus data

For example, suppose we are testing a function that uses the quadratic formula to determine the two roots of a second-degree polynomial ax^2+bx+c. For simplicity, assume that we are going to work only with real numbers, and print an error message if it turns out that the two roots are complex numbers (numbers involving the square root of a negative number). We can come up with test data for each of the four cases, based on values of the polynomial's discriminant (b^2-4ac):

Easy data (discriminant is a perfect square):

a	b	c	Roots
1	2	1	-1, -1
1	3	2	-1, -2

Typical data (discriminant is positive):

a	b	c	Roots
1	4	1	-3.73205, -0.267949
2	4	1	-1.70711, -0.292893

Boundary / extreme data (discriminant is zero):

a	b	c	Roots
2	-4	2	1, 1
2	-8	8	2, 2

Bogus data (discriminant is negative, or **a** is zero):

a	b	c	Roots
1	1	1	square root of negative number
0	1	1	division by zero

As with glass-box testing, you should test your code with each set of test data. If the answers match, then your code passes the black-box test.

What is Black Box Testing?

Firstly we will learn what is Black Box Testing? Here we will discuss about how black box testing is perform, different BBT Techniques used in testing.

Black Box Testing Method:

Black box testing is the Software testing method which is used to test the software without knowing the internal structure of code or program. Most likely this testing method is what most of tester actual perform and used the majority in the practical life.

Basically software under test is called as “Black-Box”, we are treating this as black box & without checking internal structure of software we test the software. All testing is done as customer’s point of view and tester is only aware of what is software is suppose to do but how these requests are processing by software is not aware. While testing tester is knows about the input and expected output’s of the software and they do not aware of how the software or application actually processing the input requests & giving the outputs. Tester only passes valid as well as invalid inputs & determines the correct expected outputs. All the test cases to test using such method are calculated based on requirements & specifications document.

The main purpose of the Black Box is to check whether the software is working as per expected in requirement document & whether it is meeting the user expectations or not.

There are different types of testing used in industry. Each testing type is having its own advantages & disadvantages. So fewer bugs cannot be find using the black box testing or white box testing.

Types of Black Box Testing Techniques:

Following black box testing techniques are used for testing the software application.

- Boundary Value Analysis (BVA)
- Equivalence Class Partitioning
- Decision Table based testing
- Cause-Effect Graphing Technique
- Error Guessing

1) Boundary Value Analysis (BVA):

Boundary Value Analysis is the most commonly used test case design method for black box testing. As all we know the most of errors occurs at boundary of the input values. This is one of the techniques used to find the error in the boundaries of input values rather than the center of the input value range. Boundary Value Analysis is the next step of the Equivalence class in which all test cases are design at the boundary of the Equivalence class. Let us take an example to explain this:

Suppose we have software application which accepts the input value text box ranging from 1 to 1000, in this case we have invalid and valid inputs:

Invalid Input	Valid Input	Invalid Input
0 – less	1 – 1000	1001 – above

Here are the Test cases for input box accepting numbers using Boundary value analysis:

Min value – 1	0
Min Value	1
Min value + 1	2
Normal Value	1 – 1000
Max value – 1	999
Max value	1000

Max value +1	1001

This is testing techniques is not applicable only if input value range is not fixed i.e. the boundary of input is not fixed.

2) Equivalence Class Partitioning:

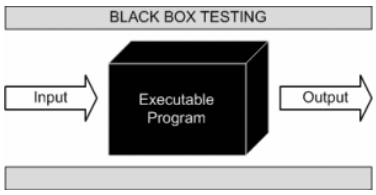
The equivalence class partition is the black box test case design technique used for writing test cases. This approach is use to reduce huge set of possible inputs to small but equally effective inputs. This is done by dividing inputs into the classes and gets one value from each class. Such method is used when exhaustive testing is most wanted & to avoid the redundancy of inputs.

In the equivalence partitioning input are divided based on the input values:

- If input value is Range, then we one valid equivalence class & two invalid equivalence classes.
- If input value is specific set, then we one valid equivalence class & one invalid equivalence classes.
- If input value is number, then we one valid equivalence class & two invalid equivalence classes.
- If input value is Boolean, then we one valid equivalence class & one invalid equivalence classes.

BLACK BOX TESTING Fundamentals:

Black Box Testing, also known as Behavioral Testing, is a software testing method in which the internal structure/ design/ implementation of the item being tested is not known to the tester. These tests can be functional or non-functional, though usually functional.



This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see.

This method attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavior or performance errors
- Initialization and termination errors

Definition by ISTQB

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

Example

A tester, without knowledge of the internal structures of a website, tests the web pages by using a browser; providing inputs (clicks, keystrokes) and verifying the outputs against the expected outcome.

Levels applicable to

Black Box testing method is applicable to the following levels of software testing:

- Integration Testing
- System Testing
- Acceptance Testing

The higher the level, and hence the bigger and more complex the box, the more black box testing method comes into use.

BLACK BOX Testing Techniques:

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

- ***Equivalence partitioning:*** It is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data.
- ***Boundary Value Analysis:*** It is a software test design technique that involves determination of boundaries for input values and selecting values that are at the boundaries and just inside/ outside of the boundaries as test data.
- ***Cause Effect Graphing:*** It is a software test design technique that involves identifying the cases (input conditions) and effects (output conditions), producing a Cause-Effect Graph, and generating test cases accordingly.

BLACK BOX Testing Advantages:

- Tests are done from a user's point of view and will help in exposing discrepancies in the specifications.
- Tester need not know programming languages or how the software has been implemented.
- Tests can be conducted by a body independent from the developers, allowing for an objective perspective and the avoidance of developer-bias.
- Test cases can be designed as soon as the specifications are complete.

BLACK BOX Testing Disadvantages:

- Only a small number of possible inputs can be tested and many program paths will be left untested.
- Without clear specifications, which are the situation in many projects, test cases will be difficult to design.
- Tests can be redundant if the software designer/ developer has already run a test case.
- Ever wondered why a soothsayer closes the eyes when foretelling events? So is almost the case in Black Box Testing.
- Black Box Testing is contrasted with White Box Testing. Read Differences between Black Box Testing and White Box Testing. Boundary Value Analysis and Equivalence Class Partitioning With Simple Example

Boundary value analysis and Equivalence Class Partitioning both are test case design techniques in black box testing. In this article we are covering "What is Boundary value analysis and equivalence partitioning & its simple examples".

What is Equivalence Class Partitioning?

Equivalence partitioning is a Test Case Design Technique to divide the input data of software into different equivalence data classes. Test cases are designed for equivalence data class. The equivalence partitions are frequently derived from the requirements specification for input data that influence the processing of the test object. A use of this method reduces the time necessary for testing software using less and effective test cases.

Equivalence Partitioning = Equivalence Class Partitioning = ECP

It can be used at any level of software for testing and is preferably a good technique to use first. In this technique, only one condition to be tested from each partition. Because we assume that, all the conditions in one partition behave in the same manner by the software. In a partition, if one condition works other will definitely work. Likewise we assume that, if one of the condition does not work then none of the conditions in that partition will work.

Equivalence partitioning is a testing technique where input values set into classes for testing.

- Valid Input Class = Keeps all valid inputs.
- Invalid Input Class = Keeps all Invalid inputs.

Example of Equivalence Class Partitioning:

- A text field permits only numeric characters
- Length must be 6-10 characters long

Partition according to the requirement should be like this:

0 1 2 3 4 5 | 6 7 8 9 10 | 11 12 13 14

Invalid | Valid | Invalid

While evaluating Equivalence partitioning, values in all partitions are equivalent that's why 0-5 are equivalent, 6 – 10 are equivalent and 11- 14 are equivalent. At the time of testing, test 4 and 12 as invalid values and 7 as valid one. It is easy to test input ranges 6–10 but harder to test input ranges 2-600. Testing will be easy in the case of lesser test cases but you should be very careful. Assuming, valid input is 7. That means, you belief that the developer coded the correct valid range (6-10).

What is Boundary value analysis?

Boundary value analysis is a test case design technique to test boundary value between partitions (both valid boundary partition and invalid boundary partition). A boundary value is an input or output value on the border of an equivalence partition, includes minimum and maximum values at inside and outside boundaries. Normally Boundary value analysis is part of stress and negative testing.

Using Boundary Value Analysis technique tester creates test cases for required input field. For example; an Address text box which allows maximum 500 characters. So, writing test cases for each character once will be very difficult so that will choose boundary value analysis.

Example for Boundary Value Analysis:

Example 1: Suppose you have very important tool at office, accept valid User Name and Password field to work on that tool, and accept minimum 8 characters and maximum 12 characters. Valid range 8-12, Invalid range 7 or less than 7 and Invalid range 13 or more than 13.

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

- Test Cases 1: Consider password length less than 8.
- Test Cases 2: Consider password of length exactly 8.
- Test Cases 3: Consider password of length between 9 and 11.
- Test Cases 4: Consider password of length exactly 12.
- Test Cases 5: Consider password of length more than 12.

Example 2: Test cases for the application whose input box accepts numbers between 1-1000. Valid range 1-1000, Invalid range 0 and Invalid range 1001 or more.

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

- Test Cases 1: Consider test data exactly as the input boundaries of input domain i.e. values 1 and 1000.
- Test Cases 2: Consider test data with values just below the extreme edges of input domains i.e. values 0 and 999.
- Test Cases 3: Consider test data with values just above the extreme edges of input domain i.e. values 2 and 1001.

Over to you on Boundary Value Analysis and Equivalence Class Partitioning:

Till now we have seen about what is Boundary value analysis and equivalence partitioning & it's simple examples. Here we have covered very basic and simple example to understand the most commonly used test case design techniques. There is no such hard and fast rule to take only one input from each partition. Based on your needs and previous experience you can decide the inputs.

ASSIGNMENT No: 3B

TITLE: Perform white box testing – Cyclomatic complexity, data flow testing, control flow testing

- **AIM:** Perform white box testing– Cyclomatic complexity, data flow testing, control flow testing
- **THEORY:**

White box and black box testing: White box testing (also known as clear, glass box or structural testing) is a testing technique which evaluates the code and internal structure of the program. It's the counterpart of Black box testing. In Black box testing, we test the software from a user's point of view, but in White box, we see and test the actual code. In Black box, we do testing without seeing the internal system code, but in White box we do see and test the internal code. White box testing technique is used by both developers as well as testers. It helps them understand which line of code is actually executed and which is not. This may indicate that there is either missing logic or a typo, which eventually can lead into some negative consequences.

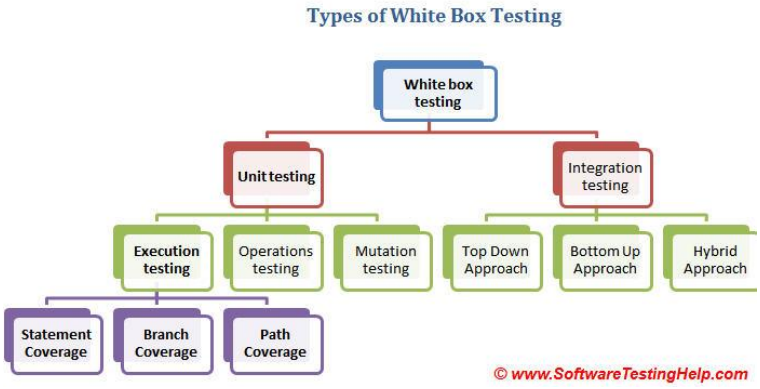
Steps to perform White box testing:

Step 1 – Understand the functionality of the application through its source code. Having said that, it simply means that the tester must be well versed with the programming language and other tools and techniques used to develop the software.

Step 2 – Create the tests and execute them. When we discuss about testing, "coverage" is the most important factor. Here I will explain how to have maximum coverage in the context of White box testing.

Types of white box testing:

There are different types & different methods for each white box testing type. See below image.



The three main White box testing Techniques are:

1. Statement Coverage
2. Branch Coverage
3. Path Coverage

1Statement coverage: In programming language, statement is nothing but the line of code or instruction for the computer to understand and act accordingly. A statement becomes an executable statement when it gets compiled and converted into the object code and performs the action when the program is in running mode. Hence "Statement Coverage", as the name suggests, is the method of validating that each and every line of code is executed at least once.

2Branch Coverage: "Branch" in programming language is like the "IF statements". If statement has two branches: true and false. So in Branch coverage (also called Decision coverage), we validate that each branch is executed at least once. In case of a "IF statement", there will be two test conditions:

- One to validate the true branch and
- Other to validate the false branch

Hence in theory, Branch Coverage is a testing method which when executed ensures that each branch from each decision point is executed.

3Path Coverage: Path coverage tests all the paths of the program. This is a comprehensive technique which ensures that all the paths of the program are traversed at least once. Path Coverage is even more powerful that Branch coverage. This technique is useful for testing the complex programs.

White box testing example:

Simple pseudo code:

INPUT A & B

C=A+B

IF C>100

PRINT"IT'S DONE"

For Statement Coverage – we would need only one test case to check all the lines of code. If I consider *TestCase_01 to be (A=40 and B=70),* then all the lines of code will be executed.

Now the question arises:

Is that sufficient?

What if I consider Test case as A=33 and B=45?

Because Statement coverage will only cover the true side, for the pseudo code, only one test case would NOT be sufficient to test it. As a tester, we have to consider the negative cases as well. Hence for

maximum coverage, we need to consider "Branch Coverage", which will evaluate the "FALSE" conditions. In real world, we may add appropriate statements when condition fails.

So now the pseudo code becomes:

INPUT A&B

C=A+B

IF C>100

PRINT "IT'S PENDING

Since statement coverage is not sufficient to test the entire pseudo code, we would require Branch coverage to ensure maximum coverage. So for Branch coverage, we would require two test cases to complete testing of this pseudo code.

TestCase_01: A=33, B=45

TestCase_02: A=25, B=30

With this, we can see that each and every line of code is executed at least once.

Here are the conclusions so far:

- Branch Coverage ensures more coverage than Statement coverage
- Branch coverage is more powerful than Statement coverage,
- 100% Branch coverage itself means 100% statement coverage,
- 100 % statement coverage does not guarantee 100% branch coverage

Path Coverage: Path coverage is used to test the complex code snippets, which basically involves loop statements or combination of loops and decision statements.

Consider this pseudo code:

INPUT A & B

C=A+B

IF C>100

PRINT"IT'S DONE"

END IF

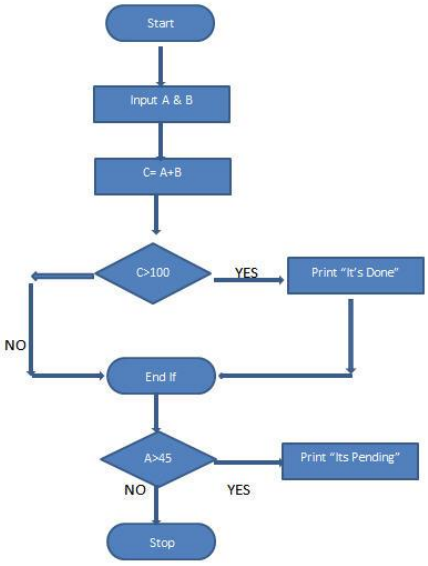
IF A>50

PRINT "IT'S PENDING

END IF

Now to ensure maximum coverage, we would require 4 test cases. There are 2 decision statements, so for each decision statement we would need to branches to test. One for true and other for false condition. So for 2 decision statements, we would require 2 test cases to test the true side and 2 test cases to test the false side, which makes total of 4 test cases.

To simplify this lets consider below flowchart of the pseudo code we have:



Path Coverage

So, In order to have the full coverage, we would need following test cases:

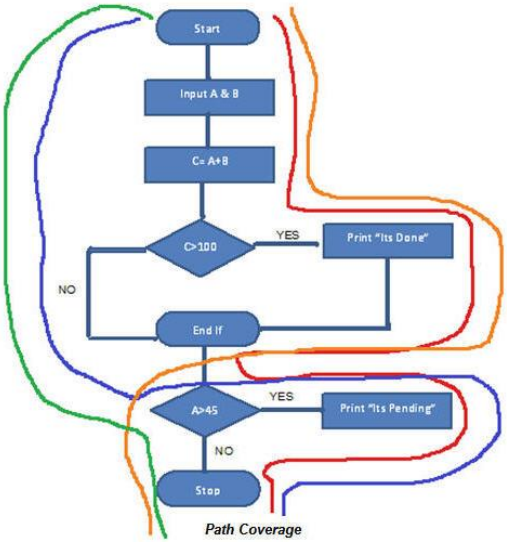
TestCase_01: A=50, B=60

TestCase_02: A=55, B=40

TestCase_03: A=40, B=65

TestCase_04: A=30, B=30

So the path covered will be:



Red Line – TestCase_01 = (A=50, B=60)

Blue Line = TestCase_02 = (A=55, B=40)

Orange Line = TestCase_03 = (A=40, B=65)

Green Line = TestCase_04 = (A=30, B=30)

Different Types of testing Conclusion:

Note that the statement, branch or path coverage does not identify any bug or defect that needs to be fixed. It only identifies those lines of code which are either never executed or remains untouched. Based on this further testing can be focused on. Relying only on black box testing is not sufficient for maximum test coverage. We need to have combination of both black box and white box testing techniques to cover maximum defects. If done properly, White box testing will certainly contribute to the software quality. It's also good for testers to participate in this testing as it can provide the most "unbiased" opinion about the code.

Cyclomatic Complexity:

- Introduced by McCabe in 1976.
- This measure provides a single ordinal number that can be compared to the complexity of their programs; it often referred to simply as program complexity.
- CC is used to measure the amount of decision logic in a single software module
- To keep software reliable, testable, manageable
- CC is based entirely on the structure of software's control flow graph
- CC used for WBT
- Used for defining the basis set of execution paths

- Basis set guarantees the execution of every statement in the program, at least once during testing.
- Minimum no. of test cases.
- One simple notation we are using Flow graphs

Software is tested from two different perspectives:

1. Internal program logic is exercised using “white box” test case design techniques.
2. Software requirements are exercised using “black box” test case design techniques.

In both cases, the intent is to find the maximum number of errors with the minimum amount of effort and time.

White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops. The "status of the program" may be examined at various points to determine if the expected or asserted status corresponds to the actual status.

White Box Testing Techniques:

White-box testing, sometimes called glass-box testing is a test case design method that uses the control structure of the procedural design to derive test cases.
Using white-box testing methods, the software engineer can derive test cases that:

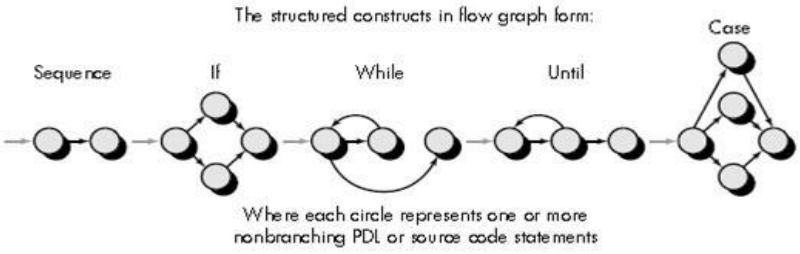
1. guarantee that all independent paths within a module have been exercised at least once
2. exercise all logical decisions on their true and false sides
3. execute all loops at their boundaries and within their operational bounds, and
4. exercise internal data structures to ensure their validity

Basis Path Testing:

Basis path testing is a white-box testing technique first proposed by Tom McCabe. The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

Flow Graph Notation

The flow graph depicts logical control flow which is used to depict the program control structure.



Referring to the figure, each circle, called a flow graph node, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct). Areas bounded by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region. Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it.

Cyclomatic Complexity / Independent Program Path:

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

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined.

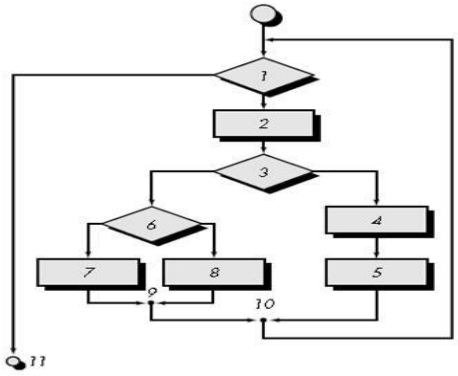


Fig A

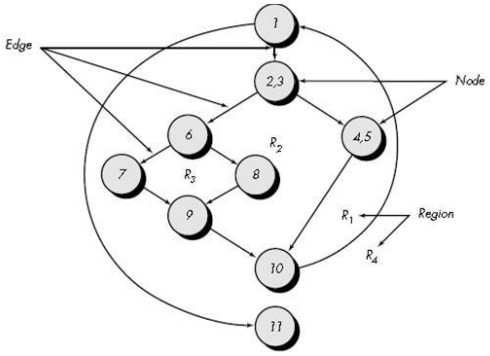


Fig B

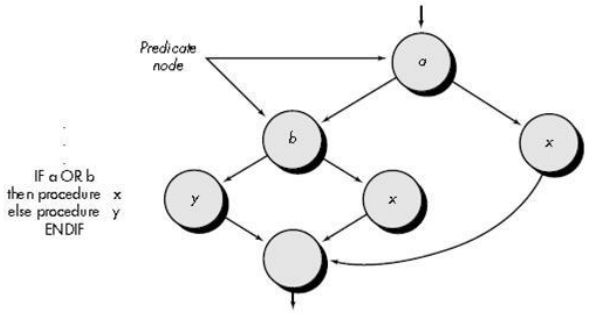


Fig C

Example:

For example, a set of independent paths for the flow graph illustrated in Figure B is:

path 1: 1-11
path 2: 1-2-3-4-5-10-1-11
path 3: 1-2-3-6-8-9-10-1-11
path 4: 1-2-3-6-7-9-10-1-11
Note that each new path introduces a new edge.

The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Paths 1, 2, 3, and 4 constitute a basis set for the flow graph in Figure B.

That is, if tests can be designed to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

How do we know how many paths to look for? The computation of Cyclomatic complexity provides the answer. Cyclomatic complexity has a foundation in graph theory and provides us with extremely useful software metric. Complexity is computed in one of three ways:

1. The number of regions of the flow graph corresponds to the Cyclomatic complexity.
2. Cyclomatic complexity, V(G), for a flow graph, G, is defined as V(G) = E - N + 2 where E is the number of flow graph edges, N is the number of flow graph nodes.
3. Cyclomatic complexity, V(G), for a flow graph, G, is also defined as V(G) = P + 1 where P is the number of predicate nodes contained in the flow graph G.

Referring once more to the flow graph in Figure B, the Cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.
2. V(G) = 11 edges - 9 nodes + 2 = 4.

3. V(G) = 3 predicate nodes + 1 = 4.

Therefore, the Cyclomatic complexity of the flow graph in Figure B is 4.

More important, the value for V(G) provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

Deriving Test Cases

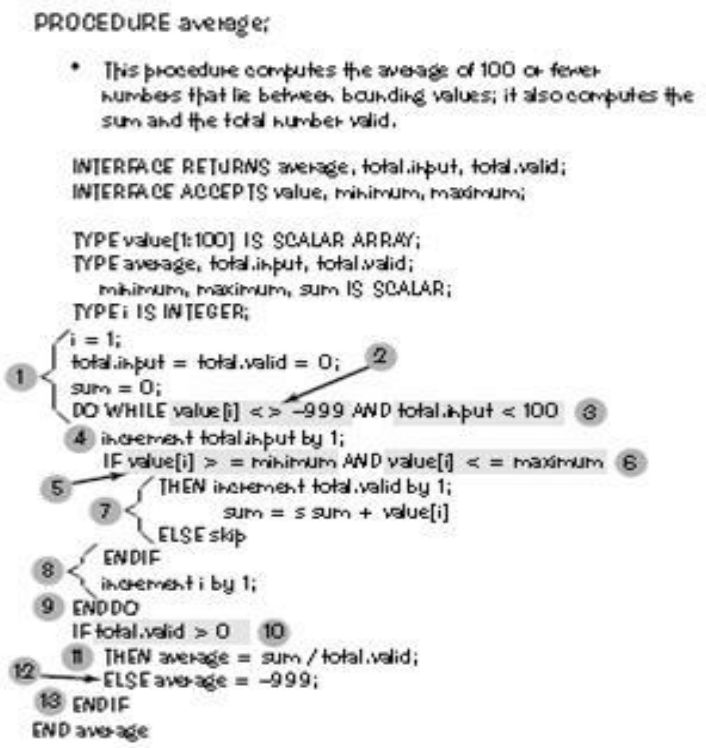


Fig D

1. Using the design or code as a foundation, draw a corresponding flow graph. A flow graph is created using the symbols and construction rules. Referring to the PDL for average in Figure A, a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes. The corresponding flow graph is in Figure B.

2. Determine the Cyclomatic complexity of the resultant flow graph. The Cyclomatic complexity, V(G), is determined. It should be noted that V(G) can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure average, compound conditions count as two) and adding 1. Referring to Figure B,

V(G) = 6 regions
V(G) = 17 edges _ 13 nodes + 2 = 6
V(G) = 5 predicate nodes + 1 = 6

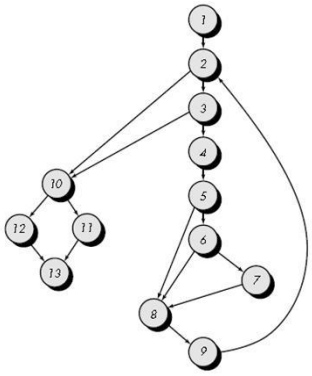


Fig E

3. Determine a basis set of linearly independent paths. The value of V(G) provides the number of linearly independent paths through the program control structure. In the case of procedure average, we expect to specify six paths:

path 1: 1-2-10-11-13
path 2: 1-2-10-12-13
path 3: 1-2-3-10-11-13
path 4: 1-2-3-4-5-8-9-2-. . .
path 5: 1-2-3-4-5-6-8-9-2-. . .
path 6: 1-2-3-4-5-6-7-8-9-2-. . .

The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable. It is often worthwhile to identify predicate nodes as an aid in the derivation of test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes.

4. Prepare test cases that will force execution of each path in the basis set. Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Test cases that satisfy the basis set just described are:

Path 1 test case:

value(k) = valid input, where k < i for 2 = i = 100
value(i) = -999 where 2 = i = 100
Expected results: Correct average based on k values and proper totals.
Note: Path 1 cannot be tested stand-alone but must be tested as part of path 4, 5, and 6 tests.

Path 2 test case:

value(1) = -999

Expected results: Average = -999; other totals at initial values.

Path 3 test case:

Attempt to process 101 or more values. First 100 values should be valid.
Expected results: Same as test case 1.

Path 4 test case:

value(i) = valid input where i < 100
value(k) < minimum where k < i
Expected results: Correct average based on k values and proper totals.

Path 5 test case:

value(i) = valid input where i < 100
value(k) > maximum where k <= i
Expected results: Correct average based on n values and proper totals.

Path 6 test case:

value(i) = valid input where i < 100

Expected results: Correct average based on n values and proper totals.

Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

It is important to note that some independent paths (e.g., path 1 in our example) cannot be tested in stand-alone fashion. That is, the combination of data required to traverse the path cannot be achieved in the normal flow of the program. In such cases, these paths are tested as part of another path test.

Note: Errors are much more common in the neighborhood of logical conditions than they are in the locus of the sequential processing. Cyclomatic complexity provides the upper bound on the number of test cases that must be executed to guarantee that every statement in a component has been executed at least once.

Usage of Cyclomatic Complexity

Risk Evaluation:

- Classification of CC and relative risk of program

Code Development Risk Analysis:

- While code is under development, it can be measured for complexity of assess inherent risk.

Test Planning:

- Mathematical analysis of CC shows the exact no. of test case to test every decision point in program.

Advantage and Disadvantage

- Measures the minimum effort and best areas of concentrations for testing.
- Measure of program's complexity and not the data complexity.

ASSIGNMENT No: 4

TITLE: Automated Testing Perform Black Box testing using automated testing tool on an application testing Points to be covered data driven wizard, parameterization, exception handling.

AIM: Automated Testing Perform Black Box testing using automated testing tool Selenium.

THEORY:-

Black Box Testing:
Black-box testing is a method of software testing that tests the functionality of an application as opposed to its internal structures or workings. Specific knowledge of the application's code/internal structure and programming knowledge in general is not required. Test cases are built around specifications and requirements, i.e., what the application is supposed to do. It uses external descriptions of the software, including specifications, requirements, and designs to derive test cases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid inputs and determines the correct output. There is no knowledge of the test object's internal structure.
This method of test can be applied to all levels of software testing: unit, integration, functional, system and acceptance. It typically comprises most if not all testing at higher levels, but can also dominate unit testing as well.

Automated Testing:
The principle of automated testing is that there is a program (which could be a job stream) that runs the program being tested, feeding it the proper input, and checking the output against the output that was expected. Once the test suite is written, no human intervention is needed, either to run the program or to look to see if it worked; the test suite does all that, and somehow indicates (say, by a :TELL message and a results file) whether the program's output was as expected. We, for instance, have over two hundred test suites, all of which can be run overnight by executing one job stream submission command; after they run, another command can show which test suites succeeded and which failed.

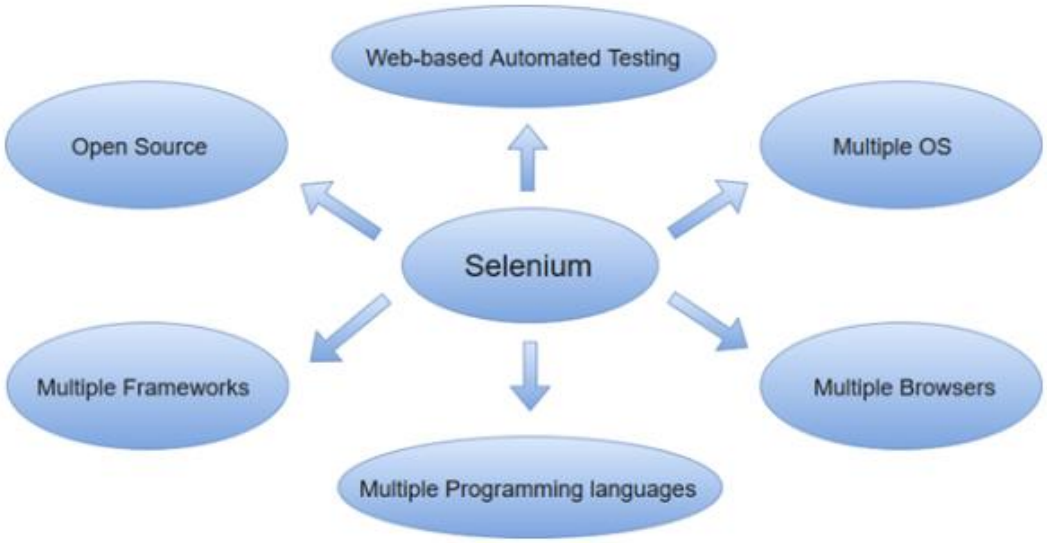
Benefits of Automated Testing:
a) Runs tests significantly faster than human users.
b) Reliable tests perform precisely the same operations each time they are run, thereby eliminating human error.
c) We can test how the software reacts under repeated execution of the same operations.
d) We can program sophisticated tests that bring out hidden information from the application.
e) We can build a suite of tests that covers every feature in your application.
f) We can reuse tests on different versions of an application, even if the user interface changes.

What is Selenium

Selenium is one of the most widely used open source Web UI (User Interface) automation testing suite. It was originally developed by Jason Huggins in 2004 as an internal tool at Thought Works. Selenium supports automation across different browsers, platforms and programming languages.

Selenium can be easily deployed on platforms such as Windows, Linux, Solaris and Macintosh. Moreover, it supports OS (Operating System) for mobile applications like IOS, windows mobile and android.

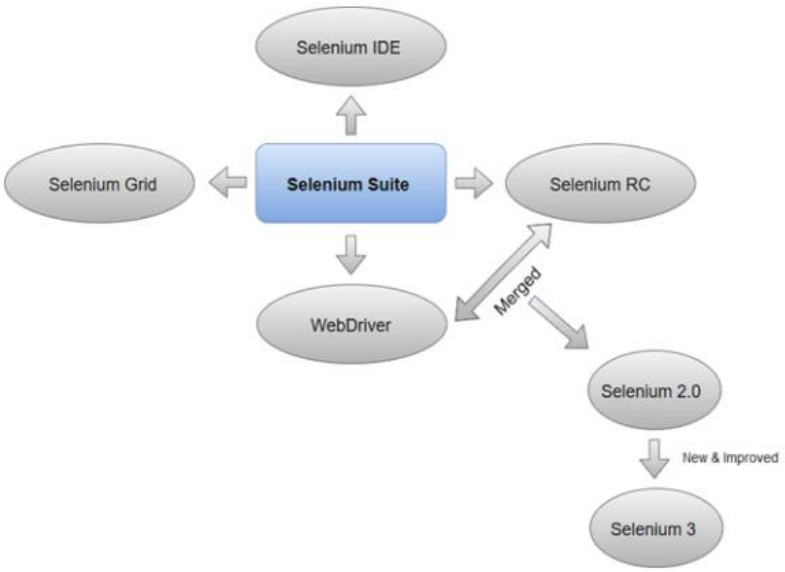
Selenium supports a variety of programming languages through the use of drivers specific to each language. Languages supported by Selenium include C#, Java, Perl, PHP, Python and Ruby. Currently, Selenium Web driver is most popular with Java and C#. Selenium test scripts can be coded in any of the supported programming languages and can be run directly in most modern web browsers. Browsers supported by Selenium include Internet Explorer, Mozilla Firefox, Google Chrome and Safari.



Selenium Tool Suite

Selenium is not just a single tool but a suite of software, each with a different approach to support automation testing. It comprises of four major components which include:

1. Selenium Integrated Development Environment (IDE)
2. Selenium Remote Control (Now Deprecated)
3. WebDriver
4. Selenium Grid



1. Selenium Integrated Development Environment (IDE)

Selenium IDE is implemented as Firefox extension which provides record and playback functionality on test scripts. It allows testers to export recorded scripts in many languages like HTML, Java, Ruby, RSpec, Python, C#, JUnit and TestNG. You can use these exported script in Selenium RC or Webdriver. Selenium IDE has limited scope and the generated test scripts are not very robust and portable.

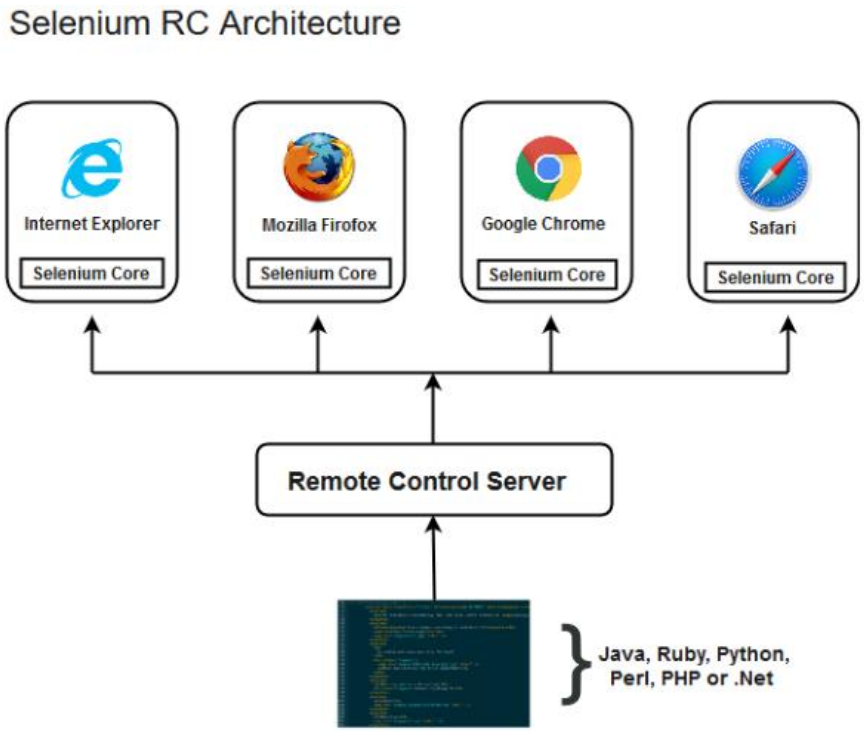
2. Selenium Remote Control

Selenium RC (officially deprecated by selenium)allows testers to write automated web application UI test in any of the supported programming languages. It also involves an HTTP proxy server which enables the browser to believe that the web application being tested comes from the domain provided by proxy server.

Selenium RC comes with two components.

1. Selenium RC Server (acts as a HTTP proxy for web requests).
2. Selenium RC Client (library containing your programming language code).

The figure given below shows the architectural representation of Selenium RC.



3. Selenium WebDriver

Selenium WebDriver (Selenium 2) is the successor to Selenium RC and is by far the most important component of Selenium Suite. Selenium WebDriver provides a programming interface to create and execute test cases. Test scripts are written in order to identify web elements on web pages and then desired actions are performed on those elements.

Selenium WebDriver performs much faster as compared to Selenium RC because it makes direct calls to the web browsers. RC on the other hand needs an RC server to interact with the web browser.

Since, WebDriver directly calls the methods of different browsers hence we have separate driver for each browser. Some of the most widely used web drivers include:

- Mozilla Firefox Driver (Gecko Driver)
- Google Chrome Driver
- Internet Explorer Driver
- Opera Driver
- Safari Driver
- HTML Unit Driver (a special headless driver)

4. Selenium Grid

Selenium Grid is also an important component of Selenium Suite which allows us to run our tests on different machines against different browsers in parallel. In simple words, we can run our tests simultaneously on different machines running different browsers and operating systems.

Selenium Grid follows the **Hub-Node Architecture** to achieve parallel execution of test scripts. The Hub is considered as master of the network and the other will be the nodes. Hub controls the execution of test scripts on various nodes of the network.

- **ASSIGNMENT No: 5**

TITLE: Defect Tracking - Prepare a Defect Tracking Report using MS-Excel

- **AIM: Defect Tracking -** Prepare a Defect Tracking Report using MS-Excel

- **Defect Repository:** is document which gives the complete details about the defects which are occurred during the testing of the software product. Defects can be classified in many ways. It is important for an organization to follow a single classification scheme and apply it to all projects. Some defects will fit into more than one class or category. Because of this problem, developers, testers, and SQA staff should try to be as consistent as possible when recording defect data. The defect types and frequency of occurrence should be used in test planning, and test design. Execution-based testing strategies should be selected that have the strongest possibility of detecting particular types of defects. The four classes of defects are as follows:

  - Requirements and specifications , Design, Code, Testing defects

1. Requirements and Specifications Defects

The beginning of the software life cycle is important for ensuring high quality in the software being developed. Defects injected in early phases can be very difficult to remove in later phases. Since many requirements documents are written using a natural language representation, they may become

  - Ambiguous, Contradictory, Unclear, Redundant, Imprecise.

Some specific requirements/specification defects are:

1.1 Functional Description Defects: The overall description of what the product does, and how it should behave (inputs/outputs), is incorrect, ambiguous, and/or incomplete.

1.2 Feature Defects: Features are described as distinguishing characteristics of a software component or system. Feature defects are due to feature descriptions that are missing, incorrect, incomplete, or unnecessary.

1.3 Feature Interaction Defects: These are due to an incorrect description of how the features should interact with each other.

1.4 Interface Description Defects: These are defects that occur in the description of how the target software is to interface with external software, hardware, and users.

2. Design Defects

Design defects occur when the following are incorrectly designed,

  - System components, Interactions between system components, Interactions between the components and outside software/hardware, or users

It includes defects in the design of algorithms, control, logic, data elements, module interface descriptions, and external software/hardware/user interface descriptions. The design defects are,

2.1 Algorithmic and Processing Defects: These occur when the processing steps in the algorithm as described by the pseudo code are incorrect.

2.2 Control, Logic, and Sequence Defects: Control defects occur when logic flow in the pseudo code is not correct.

2.3 Data Defects: These are associated with incorrect design of data structures.

2.4 Module Interface Description Defects: These defects occur because of incorrect or inconsistent usage of parameter types, incorrect number of parameters or incorrect ordering of parameters.

2.5 Functional Description Defects: The defects in this category include incorrect, missing, or unclear design elements.

2.6 External Interface Description Defects: These are derived from incorrect design descriptions for interfaces with COTS components, external software systems, databases, and hardware devices.

3. Coding Defects

Coding defects are derived from errors in implementing the code. Coding defects classes are similar to design defect classes. Some coding defects come from a failure to understand programming language constructs, and miscommunication with the designers.

3.1 Algorithmic and Processing Defects

Code related algorithm and processing defects include

- Unchecked overflow and underflow conditions, Comparing inappropriate data types, Converting one data type to another, Incorrect ordering of arithmetic operators, Misuse or omission of parentheses, Precision loss, Incorrect use of signs.

3.2 Control, Logic and Sequence Defects : This type of defects includes incorrect expression of case statements, incorrect iteration of loops, and missing paths.

3.3 Typographical Defects: These are mainly syntax errors, for example, incorrect spelling of a variable name that are usually detected by a compiler or self-reviews, or peer reviews.

3.4 Initialization Defects: This type of defects occurs when initialization statements are omitted or are incorrect. This may occur because of misunderstandings or lack of communication between programmers, or programmer`s and designer`s, carelessness, or misunderstanding of the programming environment.

3.5 Data-Flow Defects: Data-Flow defects occur when the code does not follow the necessary data-flow conditions.

3.6 Data Defects: These are indicated by incorrect implementation of data structures.

3.7 Module Interface Defects: Module Interface defects occurs because of using incorrect or inconsistent parameter types, an incorrect number of parameters, or improper ordering of the parameters.

3.8 Code Documentation Defects: When the code documentation does not describe what the program actually does, or is incomplete or ambiguous, it is called a code documentation defect.

3.9 External Hardware, Software Interfaces Defects: These defects occur because of problems related to

- System calls, Links to databases, Input/output sequences, Memory usage, Resource usage, Interrupts and exception handling, Data exchanges with hardware, Protocols, Formats, Interfaces with build files, Timing sequences.

4. Testing Defects

Test plans, test cases, test harnesses, and test procedures can also contain defects. These defects are called testing defects. Defects in test plans are best detected using review techniques.

4.1 Test Harness Defects: In order to test software, at the unit and integration levels, auxiliary code must be developed. This is called the test harness or scaffolding code. The test harness code should be carefully designed, implemented, and tested since it is a work product and this code can be reused when new releases of the software are developed.

4.2 Test Case Design and Test Procedure Defects

These consist of incorrect, incomplete, missing, inappropriate test cases, and test procedures.

Sample Defect Report

1	Defect ID	TD_library_system_Defect_01/Login
2	Defect Description	Input: Enter Invalid ID and Password Output: Login Successfully Expected :Login Unsuccessfully Defect is occurs Because of user ID or Password are wrong
3	Test Case ID	TD_library_system_Defect/Login
4	Product Version	TD_library_system_Defect/111
5	Tester	Admin
6	Build Version	Stud_ad_version0.1
7	Priority	Very High
8	Severity	High
9	Status	New
10	Reproducible	Yes, -By Valid Id and Password -By checking validation in Database
11	Reporting	Developer
12	Remark	