S.Y.M.C.A.
(TWO YEARS PATTERN)
SEMESTER - III (CBCS)

SOFTWARE TESTING & QUALITY ASSURANCE LAB

SUBJECT CODE : MCAL35

Prof. Suhas Pednekar
Vice Chancellor
University of Mumbai, Mumbai.

Prof. Ravindra D. Kulkarni
Pro Vice-Chancellor,
University of Mumbai.

Prof. Prakash Mahanwar
Director
IDOL, University of Mumbai.

Programme Co-ordinator : **Prof. Mandar Bhanushe**
Head, Faculty of Science & Technology,
IDOL, University of Mumbai - 400 098.

Course Co-ordinator : **Ms. Reshma Kurkute**
Assistant Professor B.Sc.IT, IDOL,
IDOL, University of Mumbai- 400098.

Course Writers : **Mr. Dayanand Mhamane**
Assistant Professor
G M Vedak College Of Science Tala

: **Mr. Umesh Waghmare**
Assistant Professor
K.B. Joshi Institute Of Information And Technology, Pune.

: **Mrs. Pinky Rane**
Assistant Professor
New Horizon College Of Commerce.

: **Dr. Trivani Koul**
Assistant Professor Ycc Koperkharine.

: **Dr. R. Saradha**
Assistant Professor
Sdnb Vaishnav College For Women.

: **Mrs. Laxmi Pandya**
Assistant Professor
Tolani College Of Commerce.

June 2022, Print I

Published by
Director
Institute of Distance and Open learning ,
University of Mumbai,Vidyanagari, Mumbai - 400 098.

DTP COMPOSED AND PRINTED BY
Mumbai University Press
Vidyanagari, Santacruz (E), Mumbai - 400098.

CONTENT

*****

SYLLABUS

Course Code	Course Name
MCAL35	**Software Testing & Quality Assurance Lab**

Module	Detailed Contents	Hrs
1	**Testing Basics** : Study of Review, Construction of Control Flow Graph & Writing Test Cases with case studies. Unit Testing, Integration Testing & System Testing. **Self Learning Topics:** Requirement analysis and derive test scenarios Review of Project Document, Case Study.	**4**
2	**Introduction to Selenium :** Introduction to automation Testing, Selenium latest version, Installation, Selenium WebDriver First Script. **Self Learning Topics:** Record and run a test case in Selenium IDE	**2**
3	**Selenium Web Driver Commands :** Implementing Web Drivers on Multiple Browser (chrome, Firefox),handling multiple frames Browser command, navigation Commands and find element command with Example. Locator (id, css selector, Xpath), synchronization in selenium, Handling Alerts using selenium web driver, types of alerts. Action Classes in selenium, Handling Drop Down, List Boxes, Command Button, radio buttons & text boxes. Waits command in selenium. **Self Learning Topics:** Implementation of web driver on safari	**8**
4	**TestNg Framework** : What is testNg? Installing Testng, TestNg Test, writing test cases using testNg, testNg annotation, Testing .xml **Self Learning Topics:** Parameters and dependencies from xml	**6**
5	**Automation Framework Basics :** Introduction to basic types, linear scripting, library architecture framework, data driven Framework.	**4**

	Self Learning Topics: Keyword Driven Framework	
6	**Quality Assurance** : Introduction to software quality assurance, Validation checks **Self Learning Topics:** Audits, ISO, QMSCase study	**2**

*****

MODULE I

1

TESTING BASICS

Unit Structure

1.0 STUDY OF REVIEW

All software problems can be termed as bugs. A software bug usually occurs when the software does not do what it is intended to do or does something that it is not intended to do. Flaws in specifications, design, code or other reasons can cause these bugs. Identifying and fixing bugs in the early stages of the software is very important as the cost of fixing bugs grows over time. So, the goal of a software tester is to find bugs and find them as early as possible and make sure they are fixed. Testing is context-based and risk-driven. It requires a methodical and disciplined approach to finding bugs. A good software tester needs to build credibility and possess the attitude to be explorative, troubleshooting, relentless, creative, diplomatic and persuasive. As against the perception that testing starts only after the completion of coding phase, it actually begins even before the first line of code can be written. In the life cycle of the conventional software product, testing begins at the stage when the specifications are written, i.e. from testing the product specifications or product spec. Finding bugs at this stage can save huge amounts of time and money. Once the specifications are well understood, you are required to design and execute the test cases. Selecting the appropriate technique that reduces the number of tests that cover a feature is one of the most important things that you need to take into consideration while designing these test cases. Test cases need to be designed to cover all aspects of the software, i.e. security, database, functionality (critical and general) and the user interface. Bugs originate when the test cases are executed. As a tester you might have to perform testing under different circumstances, i.e. the application could be in the initial stages or undergoing rapid changes, you

have less than enough time to test, the product might be developed using a life cycle model that does not support much of formal testing or retesting. Further, testing using different operating systems, browsers and the configurations are to be taken care of. Reporting a bug may be the most important and sometimes the most difficult task that you as a software tester will perform. By using various tools and clearly communicating to the developer, you can ensure that the bugs you find are fixed. Using automated tools to execute tests, run scripts and tracking bugs improves efficiency and effectiveness of your tests. Also, keeping pace with the latest developments in the field will augment your career as a software test engineer.

1.1 FLOW GRAPHS AND PATH TESTING

1.1.1 Flowgraphs and Path Testing:

This unit gives an in depth overview of path testing and its applications.

At the end of this unit, the student will be able to:

- Understand the concept of path testing.
- Identify the components of a control flow diagram and compare the same with a flowchart.
- Represent the control flow graph in the form of a Linked List notation.
- Understand the path testing and selection criteria and their limitations.
- Interpret a control flow-graph and demonstrate the complete path testing to achieve C1+C2.
- Classify the predicates and variables as dependant/independant and correlated/uncorrelated.
- Understand the path sensitizing method and classify whether the path is achievable or not.
- Identify the problem due to co-incidental correctness and choose a path instrumentation method to overcome the problem.

1.1.2 Basics of Path Testing:

Path Testing:

- Path Testing is the name given to a family of test techniques based on judiciously selecting a set of test paths through the program.
- If the set of paths are properly chosen then we have achieved some measure of test thoroughness. For example, pick enough paths to assure that every source statement has been executed at least once.
- Path testing techniques are the oldest of all structural test techniques.

- Path testing is most applicable to new software for unit testing. It is a structural technique.
- It requires complete knowledge of the program's structure.
- It is most often used by programmers to unit test their own code.
- The effectiveness of path testing rapidly deteriorates as the size of the software aggregate under test increases.

The Bug Assumption:

- The bug assumption for the path testing strategies is that something has gone wrong with the software that makes it take a different path than intended.
- As an example "GOTO X" where "GOTO Y" had been intended.
- Structured programming languages prevent many of the bugs targeted by path testing: as a consequence the effectiveness for path testing for these languages is reduced and for old code in COBOL, ALP, FORTRAN and Basic, the path testing is indespensable.

Control Flow Graphs:

- The control flow graph is a graphical representation of a program's control structure. It uses the elements named process blocks, decisions, and junctions.
- The flow graph is similar to the earlier flowchart, with which it is not to be confused.
- **Flow Graph Elements:** A flow graph contains four different types of elements.

(1) Process Block (2) Decisions (3) Junctions (4) Case Statements

1. Process Block:

- A process block is a sequence of program statements uninterrupted by either decisions or junctions.
- It is a sequence of statements such that if any one of statement of the block is executed, then all statement thereof are executed.
- Formally, a process block is a piece of straight line code of one statement or hundreds of statements.
- A process has one entry and one exit. It can consists of a single statement or instruction, a sequence of statements or instructions, a single entry/exit subroutine, a macro or function call, or a sequence of these.

2. Decisions:

- A decision is a program point at which the control flow can diverge.
- Machine language conditional branch and conditional skip instructions are examples of decisions.
- Most of the decisions are two-way but some are three way branches in control flow.

3. Case Statements:

- A case statement is a multi-way branch or decisions.
- Examples of case statement are a jump table in assembly language, and the PASCAL case statement.
- From the point of view of test design, there are no differences between Decisions and Case Statements

4. Junctions:

- A junction is a point in the program where the control flow can merge.
- Examples of junctions are: the target of a jump or skip instruction in ALP, a label that is a target of GOTO.

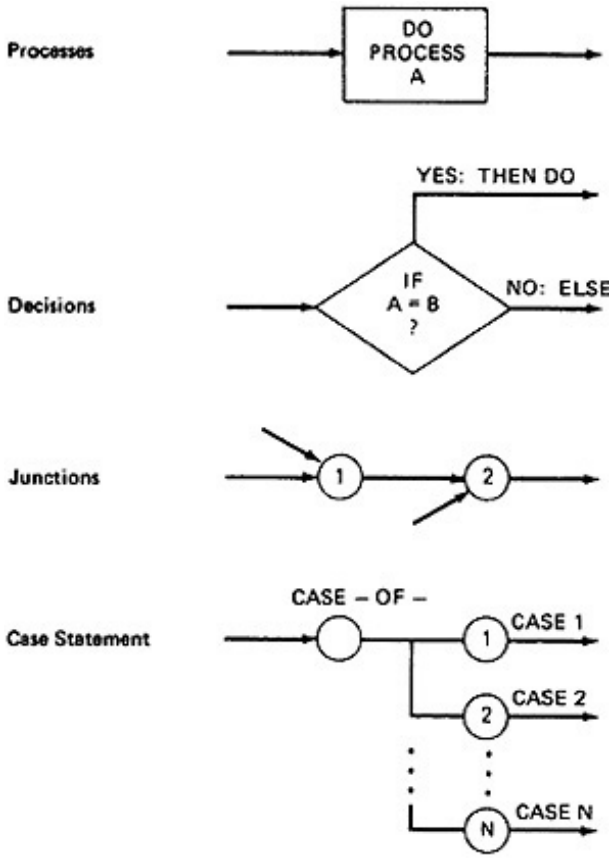


Figure 1.1: Flowgraph Elements

1.1.3 Control Flow Graphs Vs Flowcharts:

- A program's flow chart resembles a control flow graph.
- In flow graphs, we don't show the details of what is in a process block. In flow charts every part of the process block is drawn.
- The flowchart focuses on process steps, where as the flow graph focuses on control flow of the program.
- The act of drawing a control flow graph is a useful tool that can help us clarify the control flow and data flow issues.

1.1.4 Notational Evoultion:

The control flow graph is simplified representation of the program's structure. The notation changes made in creation of control flow graphs:

- The process boxes weren't really needed. There is an implied process on every line joining junctions and decisions.
- We don't need to know the specifics of the decisions, just the fact that there is a branch.
- The specific target label names aren't important-just the fact that they exist. So we can replace them by simple numbers.
- To understand this, we will go through an example (Figure 2.2) written in a FORTRAN like programming language called **Programming Design Language (PDL)**. The program's corresponding flowchart (Figure 2.3) and flowgraph (Figure 2.4) were also provided below for better understanding.
- The first step in translating the program to a flowchart is shown in Figure 2.3, where we have the typical one-for-one classical flowchart. Note that complexity has increased, clarity has decreased, and that we had to add auxiliary labels (LOOP, XX, and YY), which have no actual program counterpart. In Figure 2.4 we merged the process steps and replaced them with the single process box. We now have a control flowgraph. But this representation is still too busy. We simplify the notation further to achieve Figure 2.5, where for the first time we can really see what the control flow looks like.

CODE* (PDL)

```
      INPUT X, Y                    V(U−1):=V(U+1) + U(V−1)
      Z := X + Y                ELL:V(U+U(V)) := U + V
      V := X − Y                    IF U = V GOTO JOE
      IF Z >=Ø GOTO SAM             IF U > V THEN U := Z
JOE:  Z := Z − 1                    Z := U
SAM:  Z := Z + V                    END
      FOR U = Ø TO Z
      V(U),U(V) := (Z + V)*U
      IF V(U)= Ø GOTO JOE
      Z := Z − 1
      IF Z = Ø GOTO ELL
      U := U + 1
      NEXT U
```

* A contrived horror

Figure 1.2: Program Example (PDL)

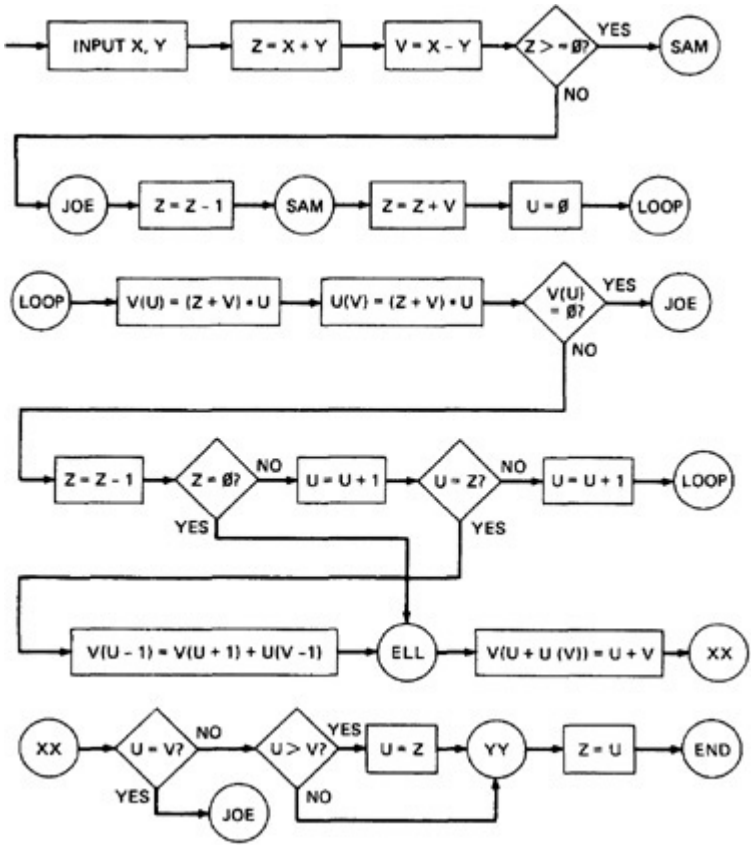


Figure 1.3: One-to-one flowchart for example program in Figure 2.2

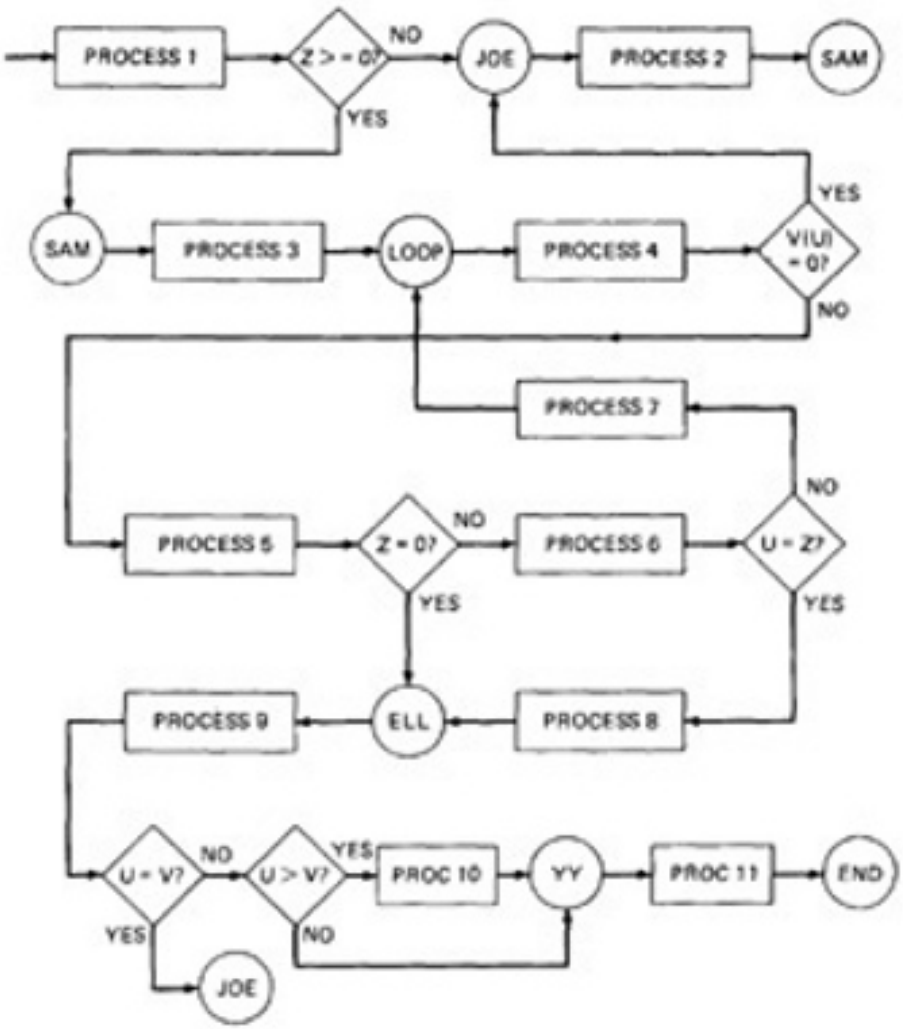


Figure 1.4: Control Flowgraph for example in Figure 1.2

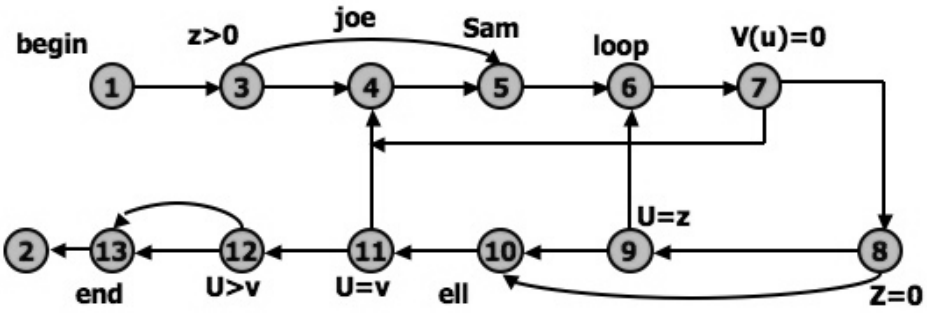


Figure 1.5: Simplified Flowgraph Notation

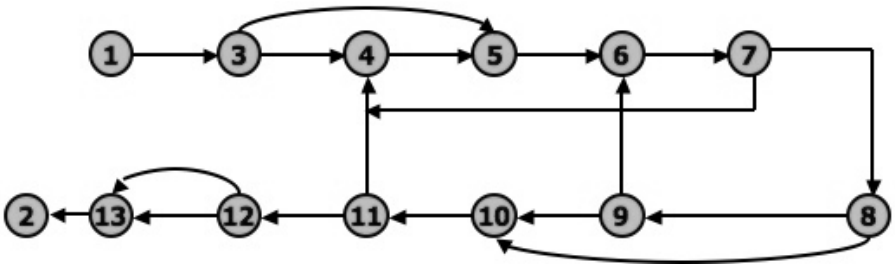


Figure 1.6: Even Simplified Flowgraph Notation

The final transformation is shown in Figure 1.6, where we've dropped the node numbers to achieve an even simpler representation. The way to work with control flowgraphs is to use the simplest possible representation - that is, no more information than you need to correlate back to the source program or PDL.

1.1.5 Linked List Representation:

Although graphical representations of flowgraphs are revealing, the details of the control flow inside a program they are often inconvenient.

In linked list representation, each node has a name and there is an entry on the list for each link in the flow graph. Only the information pertinent to the control flow is shown.

1.1.6 Linked List representation of Flow Graph:

```
 1 (BEGIN)    : 3
 2 (END)      :              Exit, no outlink
 3 (Z>Ø?)     : 4 (FALSE)
              : 5 (TRUE)
 4 (JOE)      : 5
 5 (SAM)      : 6
 6 (LOOP)     : 7
 7 (V(U)=Ø?)  : 4 (TRUE)
              : 8 (FALSE)
 8 (Z=Ø?)     : 9 (FALSE)
              :10 (TRUE)
 9 (U=Z?)     : 6 (FALSE) = LOOP
              :10 (TRUE) = ELL
10 (ELL)      :11
11 (U=V?)     : 4 (TRUE) = JOE
              :12 (FALSE)
12 (U>V?)     :13 (TRUE)
              :13 (FALSE)
13            : 2 (END)
```

Figure 1.7: Linked List Control Flowgraph Notation

Flowgraph - Program Correspondence:

A flow graph is a pictorial representation of a program and not the program itself, just as a topographic map.

You cant always associate the parts of a program in a unique way with flowgraph parts because many program structures, such as if-then-else constructs, consists of a combination of decisions, junctions, and processes.

The translation from a flowgraph element to a statement and vice versa is not always unique. (See Figure 1.8)

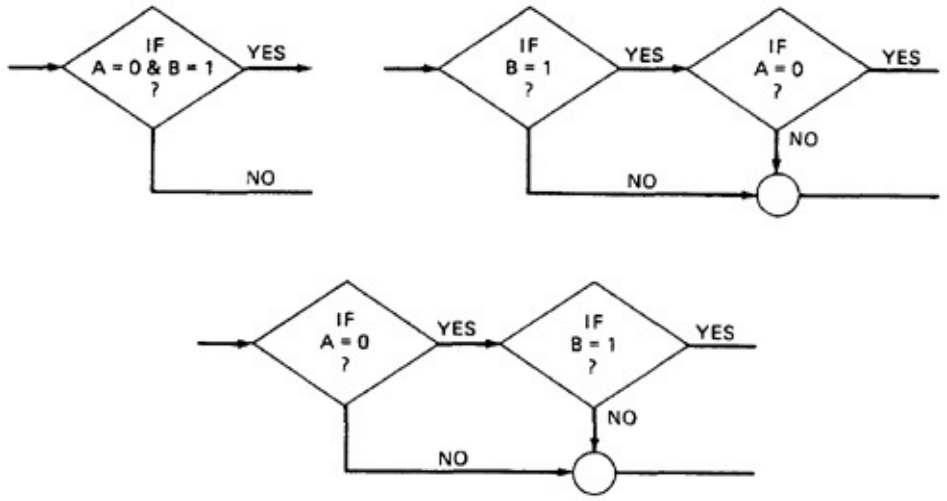


Figure 1.8: Alternative Flowgraphs for same logic (Statement "IF (A=0) AND (B=1) THEN . . .").

An improper translation from flowgraph to code during coding can lead to bugs, and improper translation during the test design lead to missing test cases and causes undiscovered bugs.

Flowgraph and Flowchart Generation:

Flowcharts can be,

1. Handwritten by the programmer.
2. Automatically produced by a flowcharting program based on a mechanical analysis of the source code.
3. Semi automatically produced by a flow charting program based in part on structural analysis of the source code and in part on directions given by the programmer.

There are relatively few control flow graph generators.

Path Testing - Paths, Nodes and Links:

Path: A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same junction, decision, or exit.

A path may go through several junctions, processes, or decisions, one or more times. Paths consists of segments.

The segment is a link - a single process that lies between two nodes.

A path segment is succession of consecutive links that belongs to some path.

The length of path measured by the number of links in it and not by the number of the instructions or statements executed along that path.

The name of a path is the name of the nodes along the path.

Fundamental Path Selection Criteria:

There are many paths between the entry and exit of a typical routine.

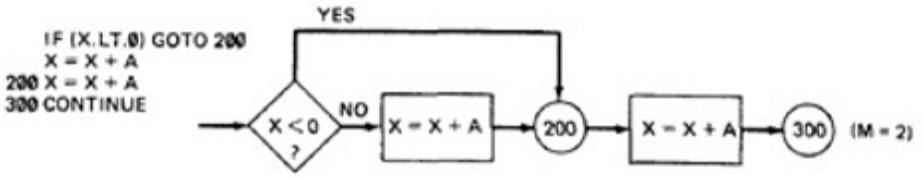
Every decision doubles the number of potential paths. And every loop multiplies the number of potential paths by the number of different iteration values possible for the loop.

Defining complete testing:

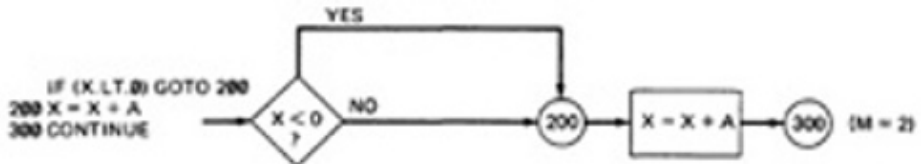
1. Exercise every path from entry to exit
2. Exercise every statement or instruction at least once
3. Exercise every branch and case statement, in each direction at least once

If prescription 1 is followed then 2 and 3 are automatically followed. But it is impractical for most routines. It can be done for the routines that have no loops, in which it is equivalent to 2 and 3 prescriptions.

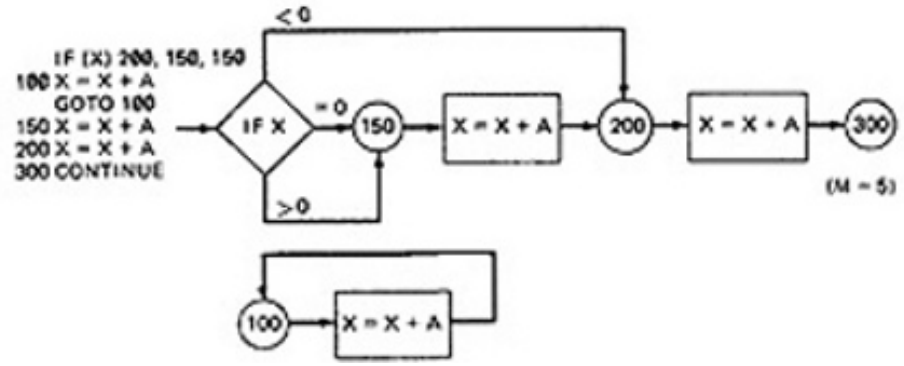
EXAMPLE: Here is the correct version.



For X negative, the output is X + A, while for X greater than or equal to zero, the output is X + 2A. Following prescription 2 and executing every statement, but not every branch, would not reveal the bug in the following incorrect version:



A negative value produces the correct answer. Every statement can be executed, but if the test cases do not force each branch to be taken, the bug can remain hidden. The next example uses a test based on executing each branch but does not force the execution of all statements:



The hidden loop around label 100 is not revealed by tests based on prescription 3 alone because no test forces the execution of statement 100 and the following GOTO statement. Furthermore, label 100 is not flagged by the compiler as an unreferenced label and the subsequent GOTO does not refer to an undefined label.

A **Static Analysis** (that is, an analysis based on examining the source code or structure) cannot determine whether a piece of code is or is not reachable. There could be subroutine calls with parameters that are subroutine labels, or in the above example there could be a GOTO that targeted label 100 but could never achieve a value that would send the program to that label.

Only a **Dynamic Analysis** (that is, an analysis based on the code's behavior while running - which is to say, to all intents and purposes, testing) can determine whether code is reachable or not and therefore distinguish between the ideal structure we think we have and the actual, buggy structure.

Path Testing Criteria:

Any testing strategy based on paths must at least both exercise every instruction and take branches in all directions.

A set of tests that does this is not complete in an absolute sense, but it is complete in the sense that anything less must leave something untested.

So we have explored three different testing criteria or strategies out of a potentially infinite family of strategies.

1. Path Testing (P_{inf}):

- Execute all possible control flow paths through the program: typically, this is restricted to all possible entry/exit paths through the program.
- If we achieve this prescription, we are said to have achieved 100% path coverage. This is the strongest criterion in the path testing strategy family: it is generally impossible to achieve.

2. Statement Testing (P_1):

- Execute all statements in the program at least once under some test. If we do enough tests to achieve this, we are said to have achieved 100% statement coverage.
- An alternate equivalent characterization is to say that we have achieved 100% node coverage. We denote this by C1.
- This is the weakest criterion in the family: testing less than this for new software is unconscionable (unprincipled or can not be accepted) and should be criminalized.

3. Branch Testing (P_2):

- Execute enough tests to assure that every branch alternative has been exercised at least once under some test.
- If we do enough tests to achieve this prescription, then we have achieved 100% branch coverage.
- An alternative characterization is to say that we have achieved 100% link coverage.
- For structured software, branch testing and therefore branch coverage strictly includes statement coverage.
- We denote branch coverage by C2.

Commonsense and Strategies:

- Branch and statement coverage are accepted today as the minimum mandatory testing requirement.
- The question "why not use a judicious sampling of paths?, what is wrong with leaving some code, untested?" is ineffectual in the view of common sense and experience since: **(1.)** Not testing a piece of a code leaves a residue of bugs in the program in proportion to the size of the untested code and the probability of bugs. **(2.)** The high probability paths are always thoroughly tested if only to demonstrate that the system works properly.
- **Which paths to be tested?** You must pick enough paths to achieve C1+C2. The question of what is the fewest number of such paths is interesting to the designer of test tools that help automate the path testing, but it is not crucial to the pragmatic (practical) design of tests. It is better to make many simple paths than a few complicated paths.

Path Selection Example:

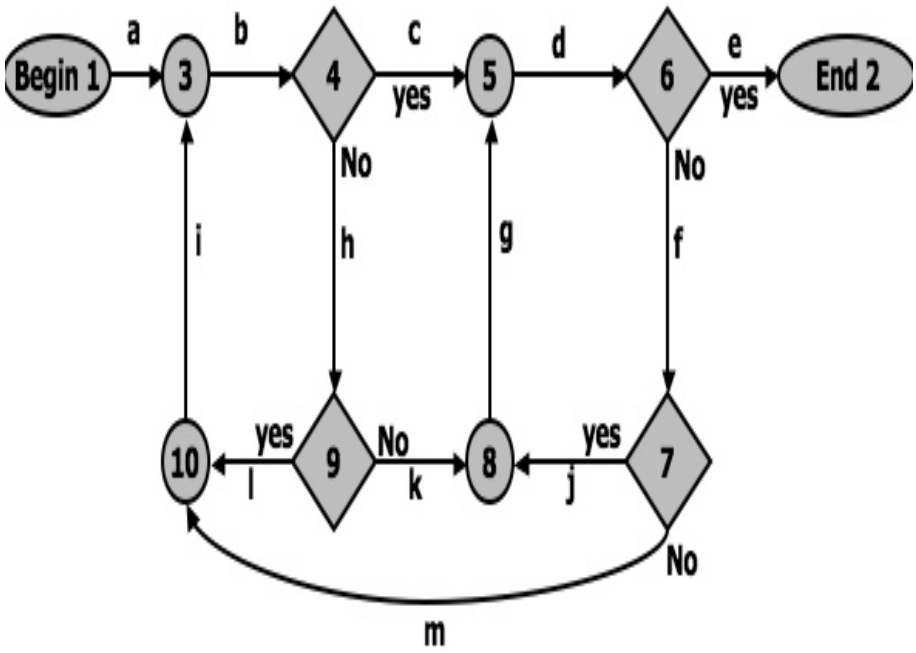


Figure 1.9: An example flowgraph to explain path selection

Practical Suggestions in Path Testing:

1. Draw the control flow graph on a single sheet of paper.
2. Make several copies - as many as you will need for coverage (C1+C2) and several more.
3. Use a yellow highlighting marker to trace paths. Copy the paths onto a master sheets.
4. Continue tracing paths until all lines on the master sheet are covered, indicating that you appear to have achieved C1+C2.
5. As you trace the paths, create a table that shows the paths, the coverage status of each process, and each decision.
6. The above paths lead to the following table considering Figure 1.9:

PATHS	DECISIONS				PROCESS-LINK											
	4	6	7	9	a	b	c	d	e	f	g	h	i	j	k	l	m
abcde	YES	YES			✓	✓	✓	✓	✓								
abhkgde	NO	YES		NO	✓	✓		✓	✓		✓	✓			✓		
abhlibcde	NO,YES	YES		YES	✓	✓	✓	✓	✓			✓	✓			✓	
abcdfjgde	YES	NO,YES	YES		✓	✓	✓	✓	✓	✓	✓			✓			
abcdfmibcde	YES	NO,YES	NO		✓	✓	✓	✓	✓	✓			✓				✓

LOOPS

7. After you have traced a a covering path set on the master sheet and filled in the table for every path, check the following:

1. Does every decision have a YES and a NO in its column? (C2)
2. Has every case of all case statements been marked? (C2)
3. Is every three - way branch (less, equal, greater) covered? (C2)
4. Is every link (process) covered at least once? (C1)

8. Revised Path Selection Rules:

- Pick the simplest, functionally sensible entry/exit path.
- Pick additional paths as small variation from previous paths. Pick paths that do not have loops rather than paths that do. Favor short paths that make sense over paths that don't.
- Pick additional paths that have no obvious functional meaning only if it's necessary to provide coverage.
- Be comfortable with your chosen paths. Play your hunches (guesses) and give your intuition free reign as long as you achieve C1+C2.
- Don't follow rules slavishly (blindly) - except for coverage.

Cases for a single loop:

A Single loop can be covered with two cases: Looping and Not looping. But, experience shows that many loop-related bugs are not discovered by C1+C2. Bugs hide themselves in corners and congregate at boundaries - in the cases of loops, at or around the minimum or maximum number of times the loop can be iterated. The minimum number of iterations is often zero, but it need not be.

CASE 1: Single loop, Zero minimum, N maximum, No excluded values:

1. Try bypassing the loop (zero iterations). If you can't, you either have a bug, or zero is not the minimum and you have the wrong case.
2. Could the loop-control variable be negative? Could it appear to specify a negative number of iterations? What happens to such a value?
3. One pass through the loop.
4. Two passes through the loop.
5. A typical number of iterations, unless covered by a previous test.
6. One less than the maximum number of iterations.
7. The maximum number of iterations.

8. Attempt one more than the maximum number of iterations. What prevents the loop-control variable from having this value? What will happen with this value if it is forced?

CASE 2: Single loop, Non-zero minimum, No excluded values:

9. Try one less than the expected minimum. What happens if the loop control variable's value is less than the minimum? What prevents the value from being less than the minimum?
10. The minimum number of iterations.
11. One more than the minimum number of iterations.
12. Once, unless covered by a previous test.
13. Twice, unless covered by a previous test.
14. A typical value.
15. One less than the maximum value.
16. The maximum number of iterations.
17. Attempt one more than the maximum number of iterations.

CASE 3: Single loops with excluded values:

- Treat single loops with excluded values as two sets of tests consisting of loops without excluded values, such as case 1 and 2 above.
- Example, the total range of the loop control variable was 1 to 20, but that values 7, 8, 9, 10 were excluded. The two sets of tests are 1-6 and 11-20.
- The test cases to attempt would be 0, 1, 2, 4, 6, 7 for the first range and 10, 11, 15, 19, 20, 21 for the second range.

Kinds of Loops:

There are only three kinds of loops with respect to path testing:

Nested Loops:

- The number of tests to be performed on nested loops will be the exponent of the tests performed on single loops.
- As we cannot always afford to test all combinations of nested loops' iterations values. Here's a tactic used to discard some of these values:

1. Start at the inner most loop. Set all the outer loops to their minimum values.
2. Test the minimum, minimum+1, typical, maximum-1 , and maximum for the innermost loop, while holding the outer loops at their minimum iteration parameter values. Expand the tests as required for out of range and excluded values.

3. If you've done the outmost loop, GOTO step 5, else move out one loop and set it up as in step 2 with all other loops set to typical values.
4. Continue outward in this manner until all loops have been covered.
5. Do all the cases for all loops in the nest simultaneously.

Concatenated Loops:

- Concatenated loops fall between single and nested loops with respect to test cases. Two loops are concatenated if it's possible to reach one after exiting the other while still on a path from entrance to exit.
- If the loops cannot be on the same path, then they are not concatenated and can be treated as individual loops.

Horrible Loops:

- A horrible loop is a combination of nested loops, the use of code that jumps into and out of loops, intersecting loops, hidden loops, and cross connected loops.
- Makes iteration value selection for test cases an awesome and ugly task, which is another reason such structures should be avoided.

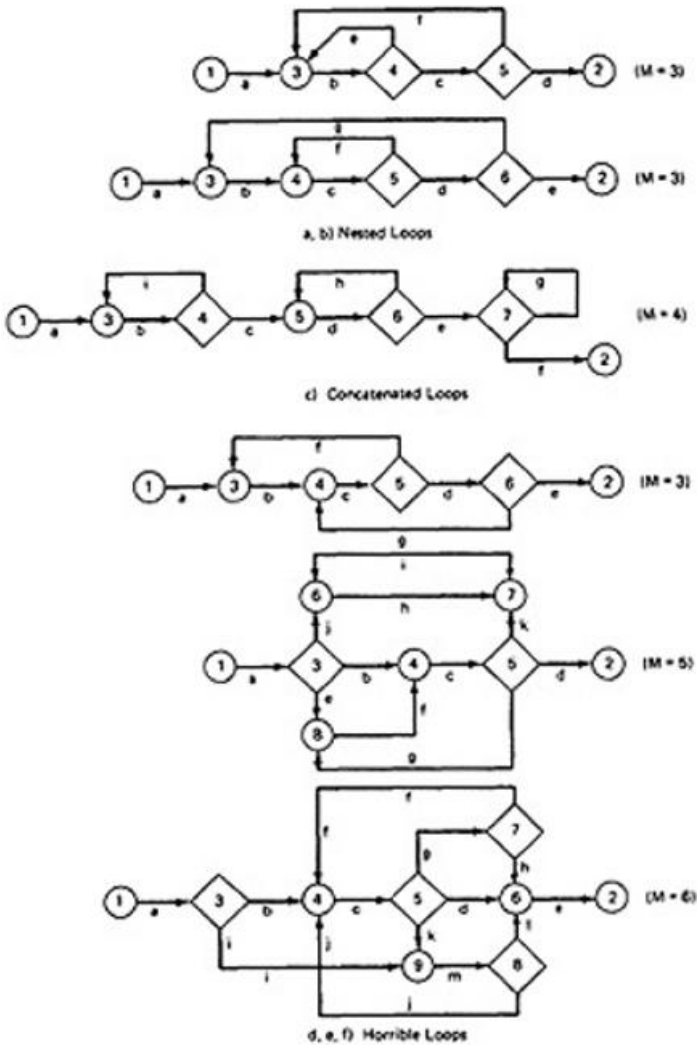


Figure 1.10: Example of Loop types

Loop Testing TIme:

- Any kind of loop can lead to long testing time, especially if all the extreme value cases are to attempted (Max-1, Max, Max+1).
- This situation is obviously worse for nested and dependent concatenated loops.
- Consider nested loops in which testing the combination of extreme values lead to long test times. Several options to deal with:
- Prove that the combined extreme cases are hypothetically possible, they are not possible in the real world
- Put in limits or checks that prevent the combined extreme cases. Then you have to test the software that implements such safety measures.

Predicates, Path Predicates and Achievable Paths:

Predicate:

The logical function evaluated at a decision is called Predicate. The direction taken at a decision depends on the value of decision variable. Some examples are: A>0, x+y>=90.....

Path Predicate:

A predicate associated with a path is called a Path Predicate. For example, "x is greater than zero", "x+y>=90", "w is either negative or equal to 10 is true" is a sequence of predicates whose truth values will cause the routine to take a specific path.

Multiway Branches:

- The path taken through a multiway branch such as a computed GOTO's, case statement, or jump tables cannot be directly expressed in TRUE/FALSE terms.
- Although, it is possible to describe such alternatives by using multi valued logic, an expedient (practical approach) is to express multiway branches as an equivalent set of if..then..else statements.
- For example a three way case statement can be written as: If case=1 DO A1 ELSE (IF Case=2 DO A2 ELSE DO A3 ENDIF)ENDIF.

Inputs:

- In testing, the word input is not restricted to direct inputs, such as variables in a subroutine call, but includes all data objects referenced by the routine whose values are fixed prior to entering it.
- For example, inputs in a calling sequence, objects in a data structure, values left in registers, or any combination of object types.

- The input for a particular test is mapped as a one dimensional array called as an Input Vector.

Predicate Interpretation:

- The simplest predicate depends only on input variables.
- For example if x1,x2 are inputs, the predicate might be x1+x2>=7, given the values of x1 and x2 the direction taken through the decision is based on the predicate is determined at input time and does not depend on processing.
- Another example, assume a predicate x1+y>=0 that along a path prior to reaching this predicate we had the assignement statement y=x2+7. Although our predicate depends on processing, we can substitute the symbolic expression for y to obtain an equivalent predicate x1+x2+7>=0.
- The act of symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called **predicate interpretation.**
- Some times the interpretation may depend on the path; for example,
- INPUT X
- ON X GOTO A, B, C, ...
- A: Z := 7 @ GOTO HEM
- B: Z := -7 @ GOTO HEM
- C: Z := 0 @ GOTO HEM
- .....
- HEM: DO SOMETHING
- .....
- HEN: IF Y + Z > 0 GOTO ELL ELSE GOTO EMM

The predicate interpretation at HEN depends on the path we took through the first multiway branch. It yields for the three cases respectively, if Y+7>0, Y-7>0, Y>0.

- The path predicates are the specific form of the predicates of the decisions along the selected path after interpretation.

Independence of Variables And Predicates:

- The path predicates take on truth values based on the values of input variables, either directly or indirectly.
- If a variable's value does not change as a result of processing, that variable is independent of the processing.

- If the variable's value can change as a result of the processing, the variable is process dependent.
- A predicate whose truth value can change as a result of the processing is said to be **process dependent** and one whose truth value does not change as a result of the processing is **process independent.**
- Process dependence of a predicate does not always follow from dependence of the input variables on which that predicate is based.

Correlation of Variables and Predicates:

- Two variables are correlated if every combination of their values cannot be independently specified.
- Variables whose values can be specified independently without restriction are called uncorrelated.
- A pair of predicates whose outcomes depend on one or more variables in common are said to be correlated predicates. For example, the predicate X==Y is followed by another predicate X+Y == 8. If we select X and Y values to satisfy the first predicate, we might have forced the 2nd predicate's truth value to change.
- Every path through a routine is achievable only if all the predicates in that routine are uncorrelated.

Path Predicate Expressions:

- A path predicate expression is a set of boolean expressions, all of which must be satisfied to achieve the selected path.
- Example:
- X1+3X2+17>=0
- X3=17
- X4-X1>=14X2
- Any set of input values that satisfy all of the conditions of the path predicate expression will force the routine to the path.
- Some times a predicate can have an OR in it.
- **Example:**

  A: X5 > 0 E: X6 < 0

  B: X1 + 3X2 + 17 >= 0 B: X1 + 3X2 + 17 >= 0

  C: X3 = 17

  D: X4 - X1 >= 14X2

  C: X3 = 17

  D: X4 - X1 >= 14X2

- Boolean algebra notation to denote the boolean expression:

ABCD+EBCD=(A+E)BCD

Predicate Coverage:

- **Compound Predicate:** Predicates of the form A OR B, A AND B and more complicated boolean expressions are called as compound predicates.
- Sometimes even a simple predicate becomes compound after interpretation. Example: the predicate if (x=17) whose opposite branch is if x.NE.17 which is equivalent to x>17 . Or. X<17.
- Predicate coverage is being the achieving of all possible combinations of truth values corresponding to the selected path have been explored under some test.
- As achieving the desired direction at a given decision could still hide bugs in the associated predicates.

Testing Blindness:

- Testing Blindness is a pathological (harmful) situation in which the desired path is achieved for the wrong reason.
- There are three types of Testing Blindness:

1. Assignment Blindness:

- Assignment blindness occurs when the buggy predicate appears to work correctly because the specific value chosen for an assignment statement works with both the correct and incorrect predicate.
- For Example:

Correct	**Buggy**
X = 7 ..... if Y > 0 then ...	X = 7 ..... if X+Y > 0 then ...

- If the test case sets Y=1 the desired path is taken in either case, but there is still a bug.

2. Equality Blindness:

- Equality blindness occurs when the path selected by a prior predicate results in a value that works both for the correct and buggy predicate.
- For Example:

Correct	Buggy
if Y = 2 then ..... if X+Y > 3 then ...	if Y = 2 then ..... if X > 1 then ...

The first predicate if y=2 forces the rest of the path, so that for any positive value of x. the path taken at the second predicate will be the same for the correct and buggy version.

3. Self Blindness:

- Self blindness occurs when the buggy predicate is a multiple of the correct predicate and as a result is indistinguishable along that path.
- For Example:

Correct	Buggy
X = A ..... if X-1 > 0 then ...	X = A ..... if X+A-2 > 0 then ...

- The assignment (x=a) makes the predicates multiples of each other, so the direction taken is the same for the correct and buggy version.

Path Sensitizing:

Review: Achievable and Unachievable Paths:

- We want to select and test enough paths to achieve a satisfactory notion of test completeness such as C1+C2.
- Extract the programs control flowgraph and select a set of tentative covering paths.
- For any path in that set, interpret the predicates along the path as needed to express them in terms of the input vector. In general individual predicates are compound or may become compound as a result of interpretation.
- Trace the path through, multiplying the individual compound predicates to achieve a boolean expression such as

(A+BC) (D+E) (FGH) (IJ) (K) (l) (L).

- Multiply out the expression to achieve a sum of products form:

ADFGHIJKL+AEFGHIJKL+BCDFGHIJKL+BCEFGHIJ KL

- Each product term denotes a set of inequalities that if solved will yield an input vector that will drive the routine along the designated path.

- Solve any one of the inequality sets for the chosen path and you have found a set of input values for the path.
- If you can find a solution, then the path is achievable.
- If you can't find a solution to any of the sets of inequalities, the path is un achievable.
- The act of finding a set of solutions to the path predicate expression is called **Path Sensitization**.

Heuristic Procedures for Sensitizing Paths:

- This is a workable approach, instead of selecting the paths without considering how to sensitize, attempt to choose a covering path set that is easy to sensitize and pick hard to sensitize paths only as you must to achieve coverage.
- Identify all variables that affect the decision.
- Classify the predicates as dependent or independent.
- Start the path selection with uncorrelated, independent predicates.
- If coverage has not been achieved using independent uncorrelated predicates, extend the path set using correlated predicates.
- If coverage has not been achieved extend the cases to those that involve dependent predicates.
- Last, use correlated, dependent predicates.

Path Instrumentation:

Path Instrumentation:

- Path instrumentation is what we have to do to confirm that the outcome was achieved by the intended path.

Co-incidental Correctness:

- The coincidental correctness stands for achieving the desired outcome for wrong reason.

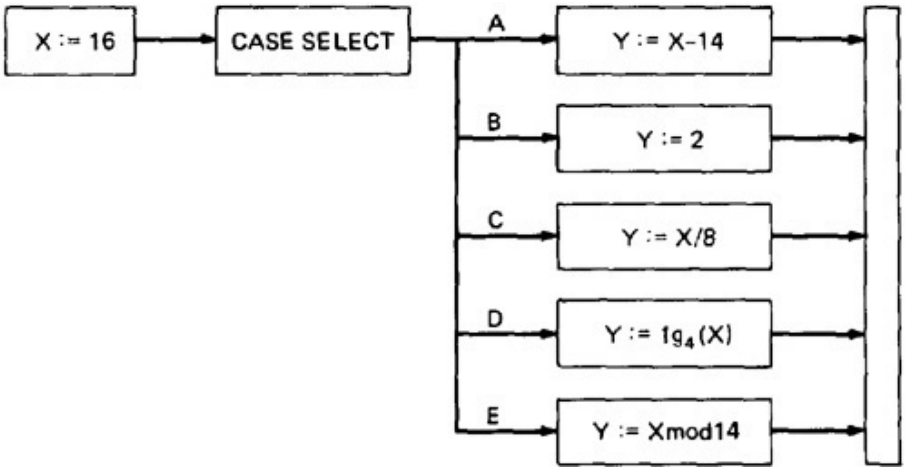


Figure 1.11: Coincidental Correctness

The above figure is an example of a routine that, for the (unfortunately) chosen input value (X = 16), yields the same outcome (Y = 2) no matter which case we select. Therefore, the tests chosen this way will not tell us whether we have achieved coverage. For example, the five cases could be totally jumbled and still the outcome would be the same. **Path Instrumentation** is what we have to do to confirm that the outcome was achieved by the intended path.

The types of instrumentation methods include:

1. Interpretive Trace Program:

- An interpretive trace program is one that executes every statement in order and records the intermediate values of all calculations, the statement labels traversed etc.
- If we run the tested routine under a trace, then we have all the information we need to confirm the outcome and, furthermore, to confirm that it was achieved by the intended path.
- The trouble with traces is that they give us far more information than we need. In fact, the typical trace program provides so much information that confirming the path from its massive output dump is more work than simulating the computer by hand to confirm the path.

2. Traversal Marker or Link Marker:

- A simple and effective form of instrumentation is called a traversal marker or link marker.
- Name every link by a lower case letter.
- Instrument the links so that the link's name is recorded when the link is executed.
- The succession of letters produced in going from the routine's entry to its exit should, if there are no bugs, exactly correspond to the path name.

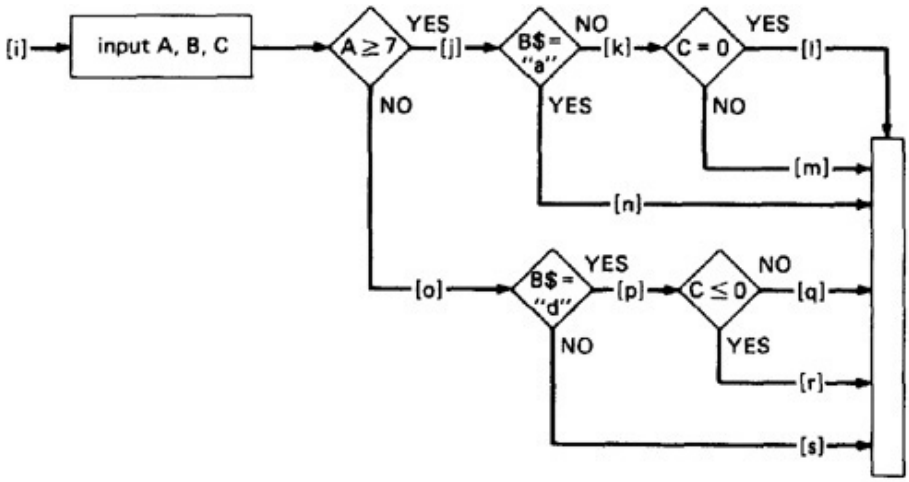


Figure 1.12: Single Link Marker Instrumentation

3. Why Single Link Markers aren't enough:

Unfortunately, a single link marker may not do the trick because links can be chewed by open bugs.

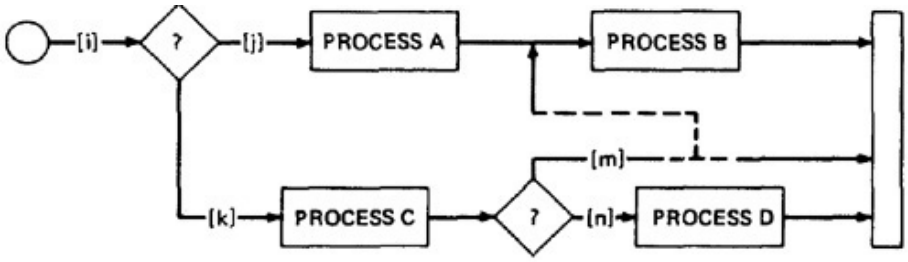


Figure 1.13: Why Single Link Markers aren't enough.

4. We intended to traverse the ikm path, but because of a rampaging GOTO in the middle of the m link, we go to process B. If coincidental correctness is against us, the outcomes will be the same and we won't know about the bug.

5. Two Link Marker Method:

- The solution to the problem of single link marker method is to implement two markers per link: one at the beginning of each link and on at the end.
- The two link markers now specify the path name and confirm both the beginning and end of the link.

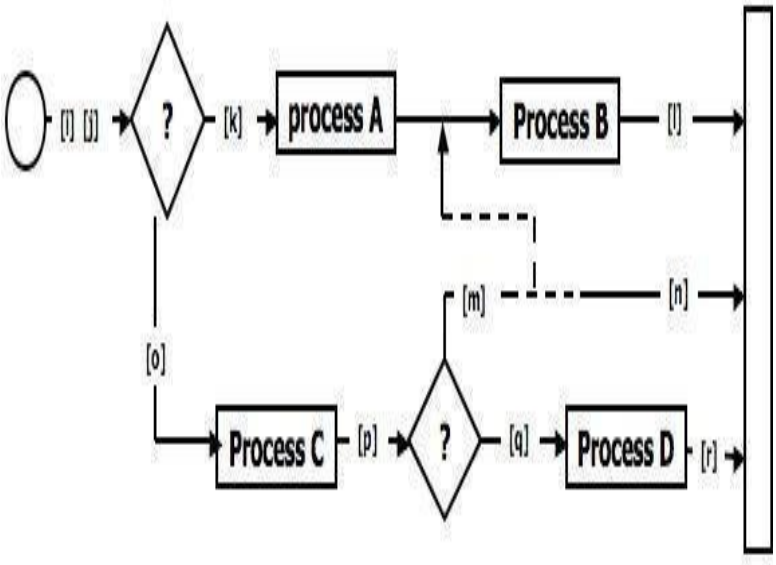


Figure 1.14: Double Link Marker Instrumentation.

6. Link Counter:

A less disruptive (and less informative) instrumentation method is based on counters. Instead of a unique link name to be pushed into a string when the link is traversed, we simply increment a link counter. We now confirm that the path length is as expected. The same problem that led us to double link markers also leads us to double link counters.

*****

2

TRANSACTION FLOW TESTING AND DATA FLOW TESTING

Unit Structure

This unit gives an indepth overview of two forms of functional or system testing namely Transaction Flow Testing and Data Flow Testing.

2.0 OBJECTIVES

At the end of this unit, the student will be able to:

- Understand the concept of transaction flow testing and data flow testing.
- Visualize the transaction flow and data flow in a software system.
- Understand the need and appreciate the usage of the two testing methods.
- Identify the complications in a transaction flow testing method and anomalies in data flow testing.
- Interpret the data flow anomaly state graphs and control flow grpahs and represent the state of the data objetcs.
- Understand the limitations of Static analysis in data flow testing.
- Compare and analyze various strategies of data flow testing.

2.1 TRANSACTION FLOWS

Introduction:

- A transaction is a unit of work seen from a system user's point of view.
- A transaction consists of a sequence of operations, some of which are performed by a system, persons or devices that are outside of the system.
- Transaction begin with Birth-that is they are created as a result of some external act.
- At the conclusion of the transaction's processing, the transaction is no longer in the system.

Example of a transaction: A transaction for an online information retrieval system might consist of the following steps or tasks:

- Accept input (tentative birth)
- Validate input (birth)
- Transmit acknowledgement to requester
- Do input processing
- Search file
- Request directions from user
- Accept input
- Validate input
- Process request
- Update file
- Transmit output
- Record transaction in log and clean up (death)

2.2 TRANSACTION FLOW GRAPHS

- Transaction flows are introduced as a representation of a system's processing.
- The methods that were applied to control flow graphs are then used for functional testing.
- Transaction flows and transaction flow testing are to the independent system tester what control flows are path testing are to the programmer.

- The transaction flow graph is to create a behavioral model of the program that leads to functional testing.
- The transaction flowgraph is a model of the structure of the system's behavior (functionality).

An example of a Transaction Flow is as follows:

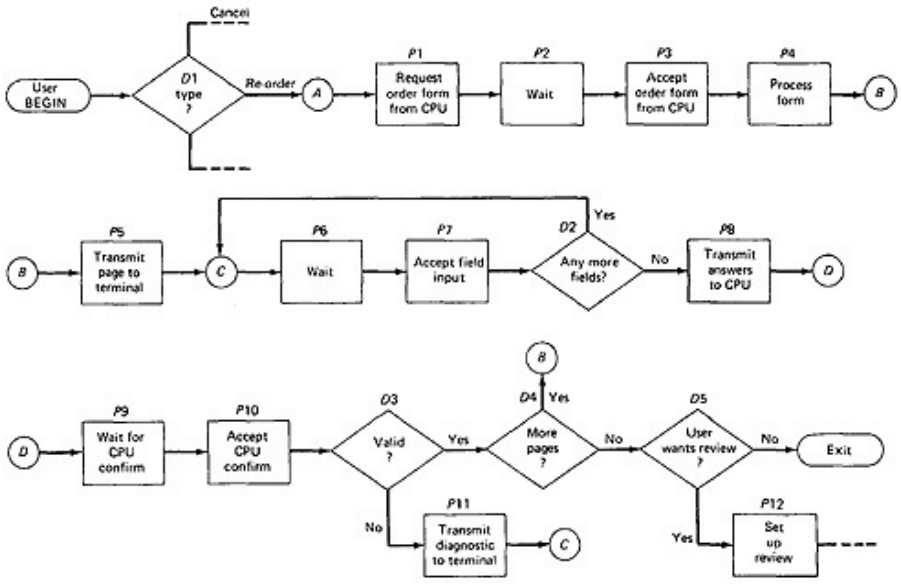


Figure 2.1: An Example of a Transaction Flow

Usage:

- Transaction flows are indispensable for specifying requirements of complicated systems, especially online systems.
- A big system such as an air traffic control or airline reservation system, has not hundreds, but thousands of different transaction flows.
- The flows are represented by relatively simple flowgraphs, many of which have a single straight-through path.
- Loops are infrequent compared to control flowgraphs.
- The most common loop is used to request a retry after user input errors. An ATM system, for example, allows the user to try, say three times, and will take the card away the fourth time.

Complications:

- In simple cases, the transactions have a unique identity from the time they're created to the time they're completed.
- In many systems the transactions can give birth to others, and transactions can also merge.

Births:

There are three different possible interpretations of the decision symbol, or nodes with two or more out links. It can be a Decision, Biosis or a Mitosis.

1. **Decision:** Here the transaction will take one alternative or the other alternative but not both. (See Figure 1.2 (a))

2. **Biosis:** Here the incoming transaction gives birth to a new transaction, and both transaction continue on their separate paths, and the parent retains it identity. (See Figure 2.2 (b))

3. **Mitosis:** Here the parent transaction is destroyed and two new transactions are created.(See Figure 2.2 (c))

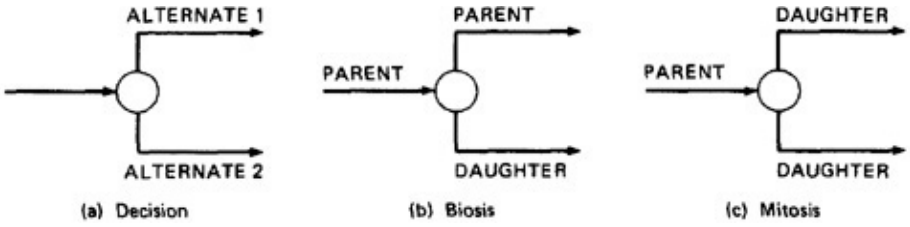


Figure 2.2: Nodes with multiple outlinks

Mergers:

Transaction flow junction points are potentially as troublesome as transaction flow splits. There are three types of junctions: (1) Ordinary Junction (2) Absorption (3) Conjugation

1. **Ordinary Junction:** An ordinary junction which is similar to the junction in a control flow graph. A transaction can arrive either on one link or the other. (See Figure 2.3 (a))

2. **Absorption:** In absorption case, the predator transaction absorbs prey transaction. The prey gone but the predator retains its identity. (See Figure 2.3 (b))

3. **Conjugation:** In conjugation case, the two parent transactions merge to form a new daughter. In keeping with the biological flavor this case is called as conjugation.(See Figure 2.3 (c))

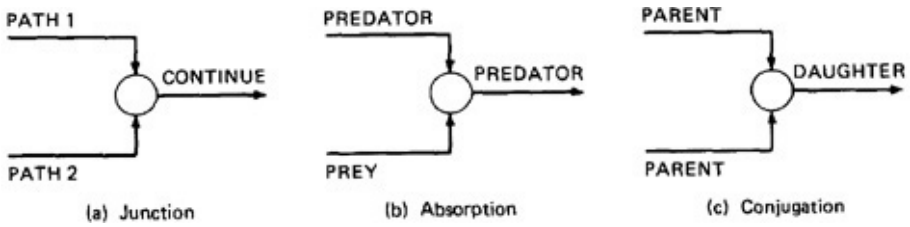


Figure 2.3: Transaction Flow Junctions and Mergers

We have no problem with ordinary decisions and junctions. Births, absorptions, and conjugations are as problematic for the software designer as they are for the software modeler and the test designer; as a

consequence, such points have more than their share of bugs. The common problems are: lost daughters, wrongful deaths, and illegitimate births.

2.3 TRANSACTION FLOW TESTING TECHNIQUES

Get The Transactions Flows:

- Complicated systems that process a lot of different, complicated transactions should have explicit representations of the transactions flows, or the equivalent.
- Transaction flows are like control flow graphs, and consequently we should expect to have them in increasing levels of detail.
- The system's design documentation should contain an overview section that details the main transaction flows.
- Detailed transaction flows are a mandatory pre requisite to the rational design of a system's functional test.

Inspections, Reviews And Walkthroughs:

- Transaction flows are natural agenda for system reviews or inspections.
- In conducting the walkthroughs, you should:
- Discuss enough transaction types to account for 98%-99% of the transaction the system is expected to process.
- Discuss paths through flows in functional rather than technical terms.
- Ask the designers to relate every flow to the specification and to show how that transaction, directly or indirectly, follows from the requirements.
- Make transaction flow testing the corner stone of system functional testing just as path testing is the corner stone of unit testing.
- Select additional flow paths for loops, extreme values, and domain boundaries.
- Design more test cases to validate all births and deaths.
- Publish and distribute the selected test paths through the transaction flows as early as possible so that they will exert the maximum beneficial effect on the project.

Path Selection:

- Select a set of covering paths (c1+c2) using the analogous criteria you used for structural path testing.

- Select a covering set of paths based on functionally sensible transactions as you would for control flow graphs.
- Try to find the most tortuous, longest, strangest path from the entry to the exit of the transaction flow.

Path Sensitization:

- Most of the normal paths are very easy to sensitize-80% - 95% transaction flow coverage (c1+c2) is usually easy to achieve.
- The remaining small percentage is often very difficult.
- Sensitization is the act of defining the transaction. If there are sensitization problems on the easy paths, then bet on either a bug in transaction flows or a design bug.

Path Instrumentation:

- Instrumentation plays a bigger role in transaction flow testing than in unit path testing.
- The information of the path taken for a given transaction must be kept with that transaction and can be recorded by a central transaction dispatcher or by the individual processing modules.
- In some systems, such traces are provided by the operating systems or a running log.

2.4 BASICS OF DATA FLOW TESTING

Data Flow Testing:

- Data flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects.
- For example, pick enough paths to assure that every data object has been initialized prior to use or that all defined objects have been used for something.
- **Motivation:** It is our belief that, just as one would not feel confident about a program without executing every statement in it as part of some test, one should not feel confident about a program without having seen the effect of using the value produced by each and every computation.

Data Flow Machines:

- There are two types of data flow machines with different architectures. (1) Von Neumann machnes (2) Multi-instruction, multi-data machines (MIMD).

- **Von Neumann Machine Architecture:**
  - Most computers today are von-neumann machines.
  - This architecture features interchangeable storage of instructions and data in the same memory units.
  - The Von Neumann machine Architecture executes one instruction at a time in the following, micro instruction sequence:
    1. Fetch instruction from memory
    2. Interpret instruction
    3. Fetch operands
    4. Process or Execute
    5. Store result
    6. Increment program counter
    7. GOTO 1
- **Multi-instruction, Multi-data machines (MIMD) Architecture:**
  - These machines can fetch several instructions and objects in parallel.
  - They can also do arithmetic and logical operations simultaneously on different data objects.
  - The decision of how to sequence them depends on the compiler.

Bug Assumption:

- The bug assumption for data-flow testing strategies is that control flow is generally correct and that something has gone wrong with the software so that data objects are not available when they should be, or silly things are being done to data objects.
- Also, if there is a control-flow problem, we expect it to have symptoms that can be detected by data-flow analysis.
- Although we'll be doing data-flow testing, we won't be using data flowgraphs as such. Rather, we'll use an ordinary control flowgraph annotated to show what happens to the data objects of interest at the moment.

Data Flow Graphs:

- The data flow graph is a graph consisting of nodes and directed links.

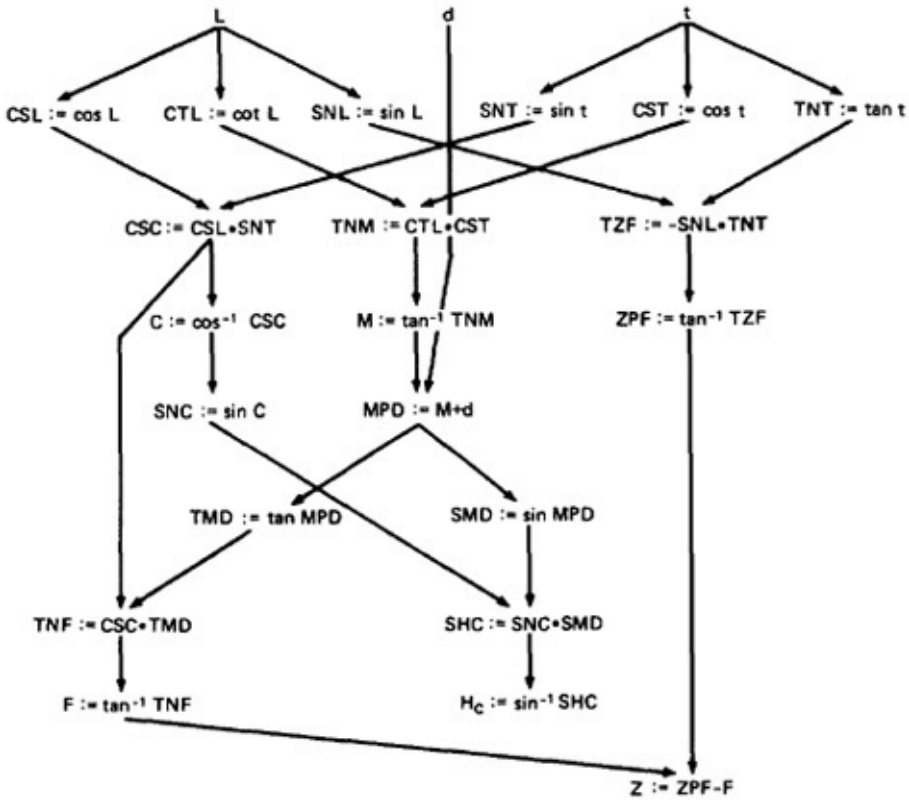


Figure 1.4: Example of a data flow graph

- We will use an control graph to show what happens to data objects of interest at that moment.
- Our objective is to expose deviations between the data flows we have and the data flows we want.

Data Object State and Usage:

- Data Objects can be created, killed and used.
- They can be used in two distinct ways: (1) In a Calculation (2) As a part of a Control Flow Predicate.
- **The following symbols denote these possibilities:**

1. **Defined:** d - defined, created, initialized etc
2. **Killed or undefined:** k - killed, undefined, released etc
3. **Usage:** u - used for something (c - used in Calculations, p - used in a predicate)

1. Defined (d):

- An object is defined explicitly when it appears in a data declaration.
- Or implicitly when it appears on the left hand side of the assignment.
- It is also to be used to mean that a file has been opened.

- A dynamically allocated object has been allocated.
- Something is pushed on to the stack.
- A record written.

2. Killed or Undefined (k):

- An object is killed on undefined when it is released or otherwise made unavailable.

3. Usage (u):

- When its contents are no longer known with certitude (with aboslute certainity / perfectness).
- Release of dynamically allocated objects back to the availability pool.
- Return of records.
- The old top of the stack after it is popped.
- An assignment statement can kill and redefine immediately. For example, if A had been previously defined and we do a new assignment such as A : = 17, we have killed A's previous value and redefined A
- A variable is used for computation (c) when it appears on the right hand side of an assignment statement.
- A file record is read or written.
- It is used in a Predicate (p) when it appears directly in a predicate.

Data Flow Anomalies:

- An anomaly is denoted by a two-character sequence of actions.
- For example, ku means that the object is killed and then used, where as dd means that the object is defined twice without an intervening usage.
- What is an anomaly is depend on the application.
- There are nine possible two-letter combinations for d, k and u. some are bugs, some are suspicious, and some are okay.

1. **dd**: probably harmless but suspicious. Why define the object twice without an intervening usage?
2. **dk**: probably a bug. Why define the object without using it?
3. **du**: the normal case. The object is defined and then used.
4. **kd**: normal situation. An object is killed and then redefined.

5. **kk**: harmless but probably buggy. Did you want to be sure it was really killed?
6. **ku**: a bug. the object doesnot exist.
7. **ud**: usually not a bug because the language permits reassignment at almost any time.
8. **uk**: normal situation.
9. **uu**: normal situation.

In addition to the two letter situations, there are six single letter situations.

We will use a leading dash to mean that nothing of interest (d,k,u) occurs prior to the action noted along the entry-exit path of interest.

A trailing dash to mean that nothing happens after the point of interest to the exit.

They possible anomalies are:

1. **-k**: possibly anomalous because from the entrance to this point on the path, the variable had not been defined. We are killing a variable that does not exist.
2. **-d**: okay. This is just the first definition along this path.
3. **-u**: possibly anomalous. Not anomalous if the variable is global and has been previously defined.
4. **k-**: not anomalous. The last thing done on this path was to kill the variable.
5. **d-**: possibly anomalous. The variable was defined and not used on this path. But this could be a global definition.
6. **u-**: not anomalous. The variable was used but not killed on this path. Although this sequence is not anomalous, it signals a frequent kind of bug. If d and k mean dynamic storage allocation and return respectively, this could be an instance in which a dynamically allocated object was not returned to the pool after use.

Data Flow Anomaly State Graph:

Data flow anomaly model prescribes that an object can be in one of four distinct states:

1. **K**: undefined, previously killed, doesnot exist
2. **D**: defined but not yet used for anything
3. **U**: has been used for computation or in predicate
4. **A**: anomalous

These capital letters (K, D, U, A) denote the state of the variable and should not be confused with the program action, denoted by lower case letters.

Unforgiving Data - Flow Anomaly Flow Graph:

Unforgiving model, in which once a variable becomes anomalous it can never return to a state of grace.

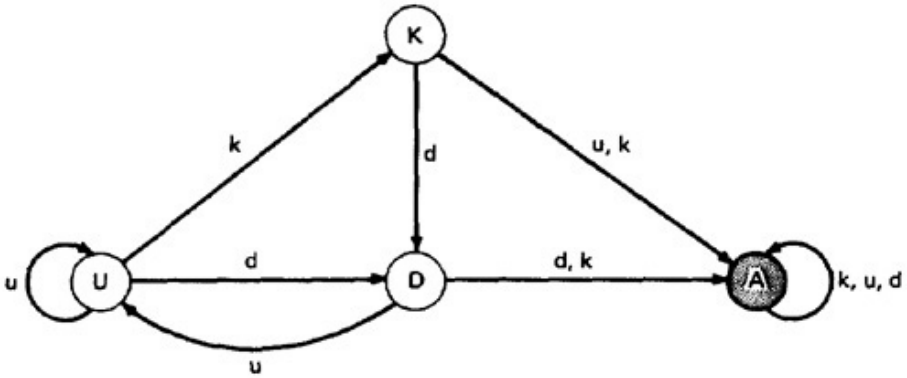


Figure 2.5: Unforgiving Data Flow Anomaly State Graph

Assume that the variable starts in the K state - that is, it has not been defined or does not exist. If an attempt is made to use it or to kill it (e.g., say that we're talking about opening, closing, and using files and that 'killing' means closing), the object's state becomes anomalous (state A) and, once it is anomalous, no action can return the variable to a working state. If it is defined (d), it goes into the D, or defined but not yet used, state. If it has been defined (D) and redefined (d) or killed without use (k), it becomes anomalous, while usage (u) brings it to the U state. If in U, redefinition (d) brings it to D, u keeps it in U, and k kills it.

Forgiving Data - Flow Anomaly Flow Graph:

Forgiving model is an alternate model where redemption (recover) from the anomalous state is possible.

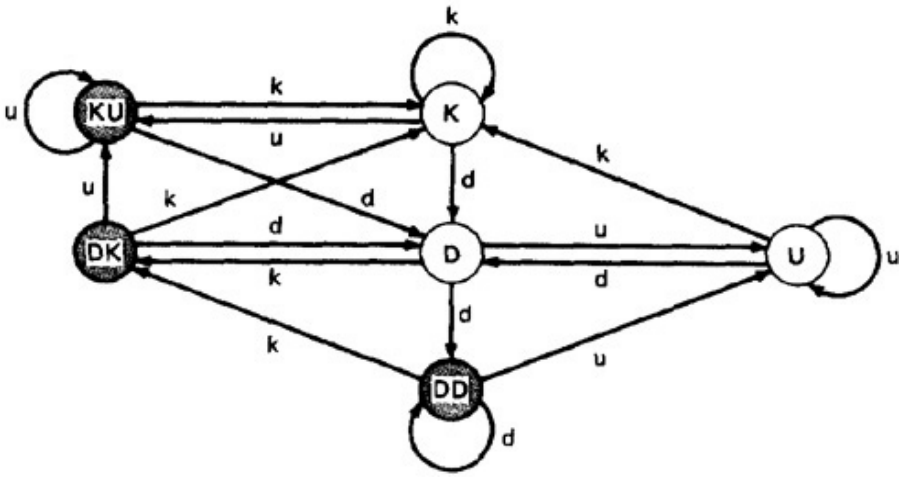


Figure 2.6: Forgiving Data Flow Anomaly State Graph

This graph has three normal and three anomalous states and he considers the kk sequence not to be anomalous. The difference between this state graph and Figure 2.5 is that redemption is possible. A proper action from any of the three anomalous states returns the variable to a useful working state.

The point of showing you this alternative anomaly state graph is to demonstrate that the specifics of an anomaly depends on such things as language, application, context, or even your frame of mind. In principle, you must create a new definition of data flow anomaly (e.g., a new state graph) in each situation. You must at least verify that the anomaly definition behind the theory or imbedded in a data flow anomaly test tool is appropriate to your situation.

2.5 STATIC VS DYNAMIC ANOMALY DETECTION

Static analysis is analysis done on source code without actually executing it. For example: source code syntax error detection is the static analysis result.

Dynamic analysis is done on the fly as the program is being executed and is based on intermediate values that result from the program's execution. For example: a division by zero warning is the dynamic result.

If a problem, such as a data flow anomaly, can be detected by static analysis methods, then it does not belongs in testing - it belongs in the language processor.

There is actually a lot more static analysis for data flow analysis for data flow anomalies going on in current language processors.

For example, language processors which force variable declarations can detect (-u) and (ku) anomalies.

But still there are many things for which current notions of static analysis are INADEQUATE.

Why Static Analysis isn't enough?:

There are many things for which current notions of static analysis are inadequate. They are:

- **Dead Variables:** Although it is often possible to prove that a variable is dead or alive at a given point in the program, the general problem is unsolvable.
- **Arrays:** Arrays are problematic in that the array is defined or killed as a single object, but reference is to specific locations within the array. Array pointers are usually dynamically calculated, so there's no way to do a static analysis to validate the pointer value. In many languages, dynamically allocated arrays contain garbage unless explicitly initialized and therefore, -u anomalies are possible.

- **Records and Pointers:** The array problem and the difficulty with pointers is a special case of multipart data structures. We have the same problem with records and the pointers to them. Also, in many applications we create files and their names dynamically and there's no way to determine, without execution, whether such objects are in the proper state on a given path or, for that matter, whether they exist at all.

- **Dynamic Subroutine and Function Names in a Call:** Subroutine or function name is a dynamic variable in a call. What is passed, or a combination of subroutine names and data objects, is constructed on a specific path. There's no way, without executing the path, to determine whether the call is correct or not.

- **False Anomalies:** Anomalies are specific to paths. Even a "clear bug" such as ku may not be a bug if the path along which the anomaly exist is unachievable. Such "anomalies" are false anomalies. Unfortunately, the problem of determining whether a path is or is not achievable is unsolvable.

- **Recoverable Anomalies and Alternate State Graphs:** What constitutes an anomaly depends on context, application, and semantics. How does the compiler know which model I have in mind? It can't because the definition of "anomaly" is not fundamental. The language processor must have a built-in anomaly definition with which you may or may not (with good reason) agree.

- **Concurrency, Interrupts, System Issues:** As soon as we get away from the simple single-task uniprocessor environment and start thinking in terms of systems, most anomaly issues become vastly more complicated. How often do we define or create data objects at an interrupt level so that they can be processed by a lower-priority routine? Interrupts can make the "correct" anomalous and the "anomalous" correct. True concurrency (as in an MIMD machine) and pseudoconcurrency (as in multiprocessing) systems can do the same to us. Much of integration and system testing is aimed at detecting data-flow anomalies that cannot be detected in the context of a single routine.

Although static analysis methods have limits, they are worth using and a continuing trend in language processor design has been better static analysis methods, especially for data flow anomaly detection. That's good because it means there's less for us to do as testers and we have far too much to do as it is.

2.6 DATA FLOW MODEL

The data flow model is based on the program's control flow graph - Don't confuse that with the program's data flowgraph.

Here we annotate each link with symbols (for example, d, k, u, c, p) or sequences of symbols (for example, dd, du, ddd) that denote the sequence

of data operations on that link with respect to the variable of interest. Such annotations are called link weights.

The control flow graph structure is same for every variable: it is the weights that change.

Components of the model:

1. To every statement there is a node, whose name is unique. Every node has at least one outlink and at least one inlink except for exit nodes and entry nodes.

2. Exit nodes are dummy nodes placed at the outgoing arrowheads of exit statements (e.g., END, RETURN), to complete the graph. Similarly, entry nodes are dummy nodes placed at entry statements (e.g., BEGIN) for the same reason.

3. The outlink of simple statements (statements with only one outlink) are weighted by the proper sequence of data-flow actions for that statement. Note that the sequence can consist of more than one letter. For example, the assignment statement A:= A + B in most languages is weighted by cd or possibly ckd for variable A. Languages that permit multiple simultaneous assignments and/or compound statements can have anomalies within the statement. The sequence must correspond to the order in which the object code will be executed for that variable.

4. Predicate nodes (e.g., IF-THEN-ELSE, DO WHILE, CASE) are weighted with the p - use(s) on every outlink, appropriate to that outlink.

5. Every sequence of simple statements (e.g., a sequence of nodes with one inlink and one outlink) can be replaced by a pair of nodes that has, as weights on the link between them, the concatenation of link weights.

6. If there are several data-flow actions on a given link for a given variable, then the weight of the link is denoted by the sequence of actions on that link for that variable.

7. Conversely, a link with several data-flow actions on it can be replaced by a succession of equivalent links, each of which has at most one data-flow action for any variable.

Let us consider the example:

CODE* (PDL)

```
        INPUT X, Y                      V(U−1):=V(U+1) + U(V−1)
        Z := X + Y                  ELL:V(U+U(V)) := U + V
        V := X − Y                      IF U = V GOTO JOE
        IF Z >=Ø GOTO SAM               IF U > V THEN U := Z
  JOE:  Z := Z − 1                      Z := U
  SAM:  Z := Z + V                      END
        FOR U = Ø TO Z
        V(U),U(V) := (Z + V)*U
        IF V(U)= Ø GOTO JOE
        Z := Z − 1
        IF Z = Ø GOTO ELL
        U := U + 1
        NEXT U
```

* A contrived horror

Figure 2.7: Program Example (PDL)

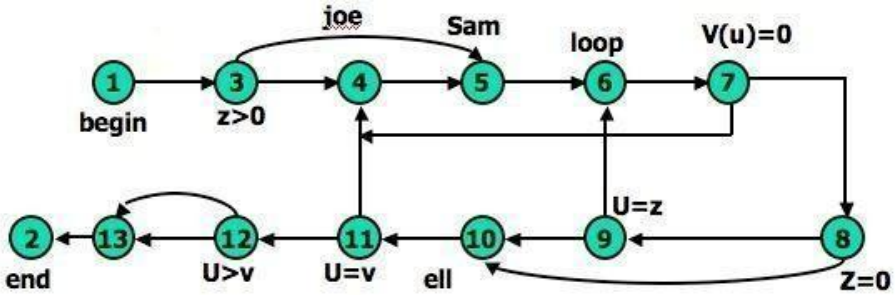


Figure 2.8: Unannotated flowgraph for example program in Figure 2.7

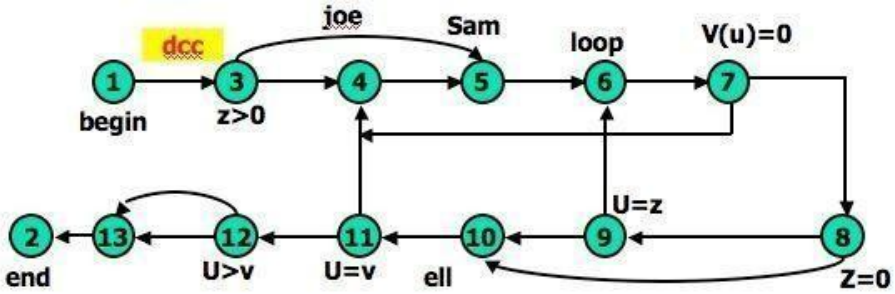


Figure 2.9: Control flowgraph annotated for X and Y data flows.

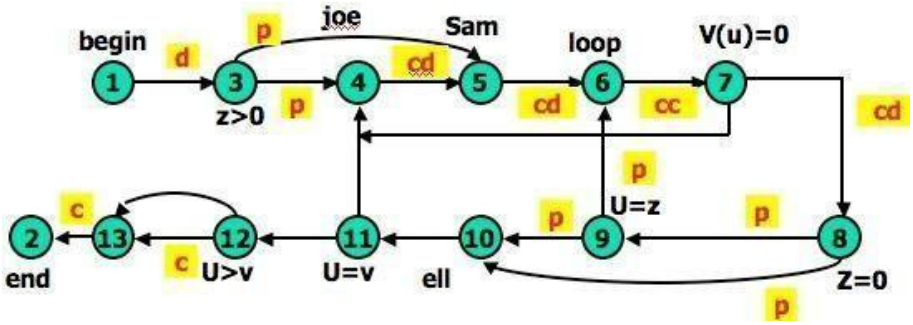


Figure 2.10: Control flowgraph annotated for Z data flow.

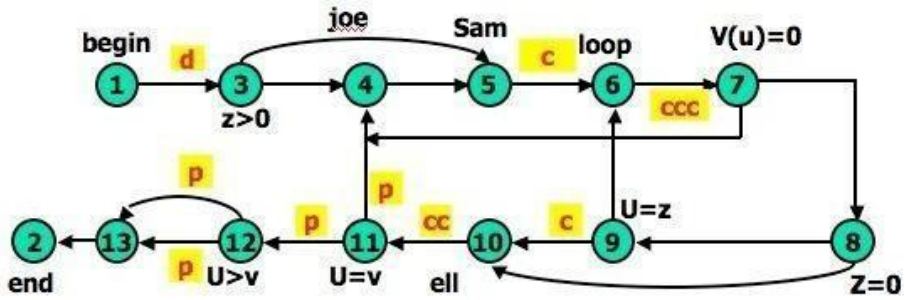


Figure 2.11: Control flowgraph annotated for V data flow.

1.7 STRATEGIES OF DATA FLOW TESTING

Introduction:

- Data Flow Testing Strategies are structural strategies.
- In contrast to the path-testing strategies, data-flow strategies take into account what happens to data objects on the links in addition to the raw connectivity of the graph.
- In other words, data flow strategies require data-flow link weights (d,k,u,c,p).
- Data Flow Testing Strategies are based on selecting test path segments (also called **sub paths**) that satisfy some characteristic of data flows for all data objects.
- For example, all subpaths that contain a d (or u, k, du, dk).
- A strategy X is **stronger** than another strategy Y if all test cases produced under Y are included in those produced under X - conversely for **weaker**.

Terminology:

1. **Definition-Clear Path Segment**, with respect to variable X, is a connected sequence of links such that X is (possibly) defined on the first link and not redefined or killed on any subsequent link of that path segment. ll paths in Figure 3.9 are definition clear because variables X and Y are defined only on the first link (1,3) and not thereafter. In Figure 3.10, we have a more complicated situation. The following path segments are definition-clear: (1,3,4), (1,3,5), (5,6,7,4), (7,8,9,6,7), (7,8,9,10), (7,8,10), (7,8,10,11). Subpath (1,3,4,5) is not definition- clear because the variable is defined on (1,3) and again on (4,5). For practice, try finding all the definition-clear subpaths for this routine (i.e., for all variables).
2. **Loop-Free Path Segment** is a path segment for which every node in it is visited atmost once. For Example, path (4,5,6,7,8,10) in Figure 3.10 is loop free, but path (10,11,4,5,6,7,8,10,11,12) is not because nodes 10 and 11 are each visited twice.

3. **Simple path segment** is a path segment in which at most one node is visited twice. For example, in Figure 3.10, (7,4,5,6,7) is a simple path segment. A simple path segment is either loop-free or if there is a loop, only one node is involved.

4. A **du path** from node i to k is a path segment such that if the last link has a computational use of X, then the path is simple and definition-clear; if the penultimate (last but one) node is j - that is, the path is (i,p,q,...,r,s,t,j,k) and link (j,k) has a predicate use - then the path from i to j is both loop-free and definition-clear.

Strategies:

The structural test strategies discussed below are based on the program's control flowgraph. They differ in the extent to which predicate uses and/or computational uses of variables are included in the test set. Various types of data flow testing strategies in decreasing order of their effectiveness are:

1. **All - du Paths (ADUP):** The all-du-paths (ADUP) strategy is the strongest data-flow testing strategy discussed here. It requires that every du path from every definition of every variable to every use of that definition be exercised under some test.

For variable X and Y: In Figure 2.9, because variables X and Y are used only on link (1,3), any test that starts at the entry satisfies this criterion (for variables X and Y, but not for all variables as required by the strategy).

For variable Z: The situation for variable Z (Figure 3.10) is more complicated because the variable is redefined in many places. For the definition on link (1,3) we must exercise paths that include subpaths (1,3,4) and (1,3,5). The definition on link (4,5) is covered by any path that includes (5,6), such as subpath (1,3,4,5,6, ...). The (5,6) definition requires paths that include subpaths (5,6,7,4) and (5,6,7,8).

For variable V: Variable V (Figure 2.11) is defined only once on link (1, 3). Because V has a predicate use at node 12 and the subsequent path to the end must be forced for both directions at node 12, the all-du-paths strategy for this variable requires that we exercise all loop-free entry/exit paths and at least one path that includes the loop caused by (11,4). Note that we must test paths that include both subpaths (3,4,5) and (3,5) even though neither of these has V definitions. They must be included because they provide alternate du paths to the V use on link (5,6). Although (7,4) is not used in the test set for variable V, it will be included in the test set that covers the predicate uses of array variable V() and U.

The all-du-paths strategy is a strong criterion, but it does not take as many tests as it might seem at first because any one test simultaneously satisfies the criterion for several definitions and uses of several different variables.

1. All Uses Startegy (AU):

The all uses strategy is that at least one definition clear path from every definition of every variable to every use of that definition be exercised under some test. Just as we reduced our ambitions by stepping down from all paths (P) to branch coverage (C2), say, we can reduce the number of test cases by asking that the test set should include at least one path segment from every definition to every use that can be reached by that definition.

For variable V: In Figure 2.11, ADUP requires that we include subpaths (3,4,5) and (3,5) in some test because subsequent uses of V, such as on link (5,6), can be reached by either alternative. In AU either (3,4,5) or (3,5) can be used to start paths, but we don't have to use both. Similarly, we can skip the (8,10) link if we've included the (8,9,10) subpath. Note the hole. We must include (8,9,10) in some test cases because that's the only way to reach the c use at link (9,10) - but suppose our bug for variable V is on link (8,10) after all? Find a covering set of paths under AU for Figure 3.11.

2. All p-uses/some c-uses strategy (APU+C):

For every variable and every definition of that variable, include at least one definition free path from the definition to every predicate use; if there are definitions of the variables that are not covered by the above prescription, then add computational use test cases as required to cover every definition.

For variable Z:

In Figure 3.10, for APU+C we can select paths that all take the upper link (12,13) and therefore we do not cover the c-use of Z: but that's okay according to the strategy's definition because every definition is covered. Links (1,3), (4,5), (5,6), and (7,8) must be included because they contain definitions for variable Z. Links (3,4), (3,5), (8,9), (8,10), (9,6), and (9,10) must be included because they contain predicate uses of Z. Find a covering set of test cases under APU+C for all variables in this example - it only takes two tests.

For variable V:

In Figure 3.11, APU+C is achieved for V by (1, 3, 5, 6, 7, 8, 10, 11, 4, 5, 6, 7, 8, 10, 11, 12[upper], 13, 2) and (1,3,5,6,7,8,10,11,12[lower], 13,2). Note that the c-us e at (9, 10) need not be included under the APU+C criterion.

3. All c-uses/some p-uses strategy (ACU+P):

The all c-uses/some p-uses strategy (ACU+P) is to first ensure coverage by computational use cases and if any definition is not covered by the previously selected paths, add such predicate use cases as are needed to assure that every definition is included in some test.

For variable Z:

In Figure 2.10, ACU+P coverage is achieved for Z by path (1,3,4,5,6,7,8,10, 11,12,13[lower], 2), but the predicate uses of several definitions are not covered. Specifically, the (1,3) definition is not covered for the (3,5) p-use, the (7,8) definition is not covered for the (8,9), (9,6) and (9, 10) p-uses.

The above examples imply that APU+C is stronger than branch coverage but ACU+P may be weaker than, or incomparable to, branch coverage.

4. All Definitions Strategy (AD):

The all definitions strategy asks only every definition of every variable be covered by atleast one use of that variable, be that use a computational use or apredicate use.

For variable Z:

Path (1,3,4,5,6,7,8, . . .) satisfies this criterion for variable Z, whereas any entry/exit path satisfies it for variable V.

From the definition of this strategy we would expect it to be weaker than both ACU+P and APU+C.

5. All Predicate Uses (APU), All Computational Uses (ACU) Strategies:

The all predicate uses strategy is derived from APU+C strategy by dropping the requirement that we include a c-use for the variable if there are no p-uses for the variable. The all computational uses strategy is derived from ACU+P strategy by dropping the requirement that we include a p- use for the variable if there are no c-uses for the variable.

It is intuitively obvious that ACU should be weaker than ACU+P and that APU should be weaker than APU+C.

Ordering The Strategies:

- Figure 2.12 compares path-flow and data-flow testing strategies. The arrows denote that the strategy at the arrow's tail is stronger than the strategy at the arrow's head.

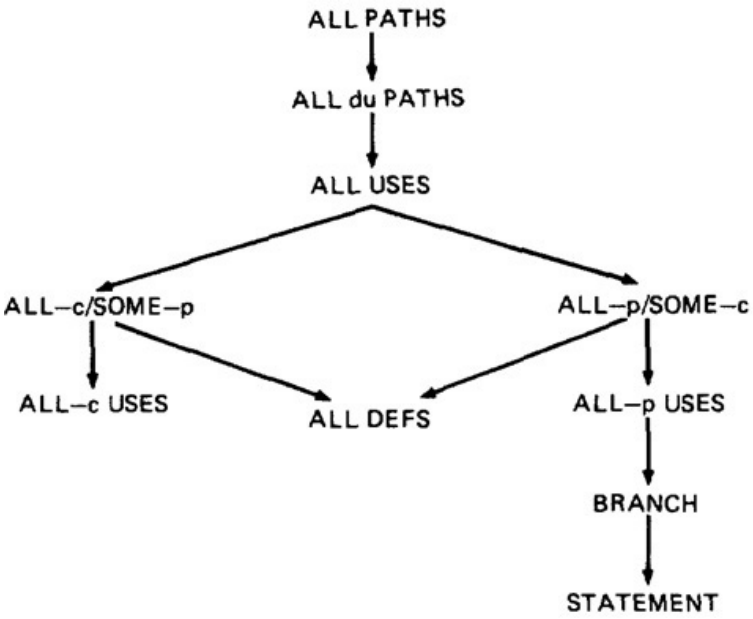


Figure 2.12: Relative Strength of Structural Test Strategies

- The right-hand side of this graph, along the path from "all paths" to "all statements" is the more interesting hierarchy for practical applications.
- Note that although ACU+P is stronger than ACU, both are incomparable to the predicate-biased strategies. Note also that "all definitions" is not comparable to ACU or APU.

Slicing And Dicing:

- A (static) program **slice** is a part of a program (e.g., a selected set of statements) defined with respect to a given variable X (where X is a simple variable or a data vector) and a statement i: it is the set of all statements that could (potentially, under static analysis) affect the value of X at statement i - where the influence of a faulty statement could result from an improper computational use or predicate use of some other variables at prior statements.
- If X is incorrect at statement i, it follows that the bug must be in the program slice for X with respect to i
- A program **dice** is a part of a slice in which all statements which are known to be correct have been removed.
- In other words, a dice is obtained from a slice by incorporating information obtained through testing or experiment (e.g., debugging).
- The debugger first limits her scope to those prior statements that could have caused the faulty value at statement i (the slice) and then eliminates from further consideration those statements that testing has shown to be correct.

- Debugging can be modeled as an iterative procedure in which slices are further refined by dicing, where the dicing information is obtained from ad hoc tests aimed primarily at eliminating possibilities. Debugging ends when the dice has been reduced to the one faulty statement.
- **Dynamic slicing** is a refinement of static slicing in which only statements on achievable paths to the statement in question are included.

2.8 UNIT TESTING

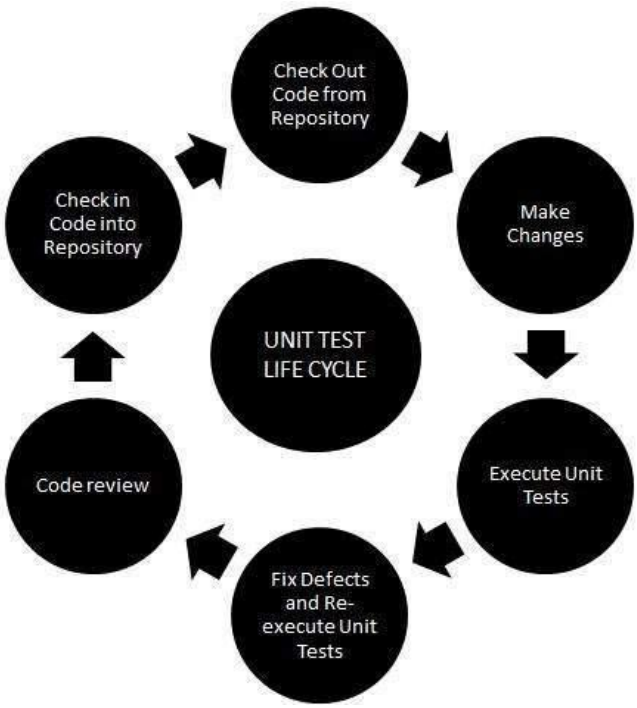
Unit testing, a testing technique using which individual modules are tested to determine if there are any issues by the developer himself. It is concerned with functional correctness of the standalone modules.

The main aim is to isolate each unit of the system to identify, analyze and fix the defects.

Unit Testing - Advantages:

- Reduces Defects in the Newly developed features or reduces bugs when changing the existing functionality.
- Reduces Cost of Testing as defects are captured in very early phase.
- Improves design and allows better refactoring of code.
- Unit Tests, when integrated with build gives the quality of the build as well.

Unit Testing LifeCyle:



Unit Testing Techniques:

- **Black Box Testing:** Using which the user interface, input and output are tested.
- **White Box Testing:** used to test each one of those functions behaviour is tested.
- **Gray Box Testing:** Used to execute tests, risks and assessment methods.

2.9 INTEGRATION TESTING

Upon completion of unit testing, the units or modules are to be integrated which gives raise to integration testing. The purpose of integration testing is to verify the functional, performance, and reliability between the modules that are integrated.

Integration Strategies:

- Big-Bang Integration
- Top Down Integration
- Bottom Up Integration
- Hybrid Integration

2.10 SYSTEM TESTING

System Testing (ST) is a black box testing technique performed to evaluate the complete system the system's compliance against specified requirements. In System testing, the functionalities of the system are tested from an end-to-end perspective.

System Testing is usually carried out by a team that is independent of the development team in order to measure the quality of the system unbiased. It includes both functional and Non-Functional testing.

*****

MODULE II

3

INTRODUCTION TO SELENIUM

Unit Structure

3.0 OBJECTIVES

- In Selenium Develop test cases to detect bugs and errors.
- Automation framework design and implementation according to project layout.
- Improvement and automated test practices.
- Participate in communicating best practices in projects.
- In Project to define test strategies and test manuals for tracking and fixing software issues.
- Describe the characteristics of Automation Testing & its methods.

3.1 INTRODUCTION

Testing is the most crucial phase in the software development lifecycle and its main objective is to ensure bug-free software that meets customer requirements. Testing is strenuous since it involves manual execution of test cases against various applications to detect bugs and errors.

But what if we could automate the testing process?

Now before we understand what Selenium is, let us focus on the challenges with manual testing.

3.2 CHALLENGES WITH MANUAL TESTING

Manual testing is one of the primitive ways of software testing. It doesn't require the knowledge of any software testing tool and can practically test any application.

The tester manually executes test cases against applications and compares the actual results with desired results. Any differences between the two are considered as defects and are immediately fixed. The tests are then re-run to ensure an utterly error-free application.

Manual testing has its own drawbacks, however, a few of which can include:

- It's extremely time-consuming
- There's a high risk of error
- It requires the presence of a tester 24/7
- Requires manual creation of logs
- Has a limited scope

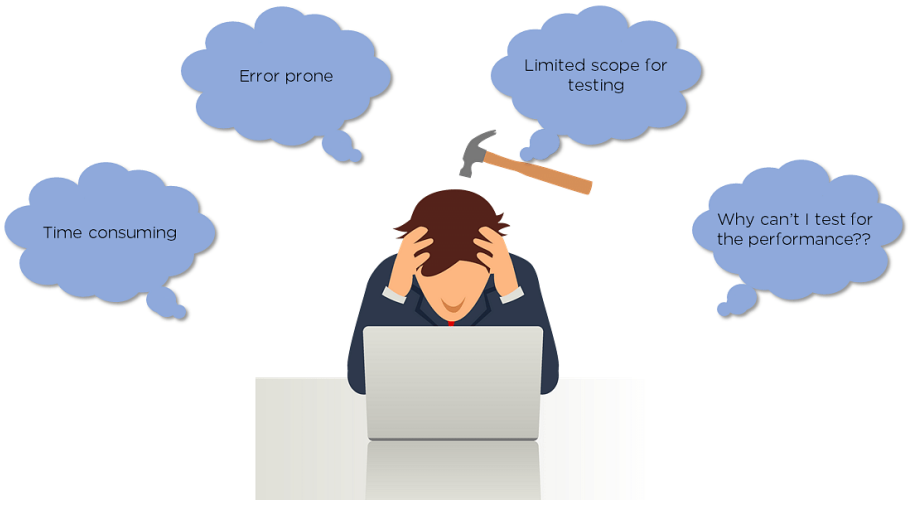


Fig: 3.2 Challenges with manual testing

Considering all the drawbacks, a desperate need to automate the testing process was in demand. Now, let us understand the advent of Selenium before looking into what Selenium is.

3.3 INVENTION OF SELENIUM

Jason Huggins, an engineer at ThoughtWorks, Chicago, found manual testing repetitive and boring. He developed a JavaScript program to automate the testing of a web application, called JavaScriptTestRunner.

Initially, the new invention was deployed by the employees at Thoughtworks. However, in 2004, it was renamed Selenium and was made open source. Since its inception, Selenium has been a powerful automation testing tool to test various web applications across different platforms.

3.4 WHAT IS SELENIUM?

Selenium is an open-source, automated testing tool used to test web applications across various browsers. Selenium can only test web applications, unfortunately, so desktop and mobile apps can't be tested.

However, other tools like Appium and HP's QTP can be used to test software and mobile applications.

3.5 IMPORTANCE OF SELENIUM TOOLS

1. Selenium is easy to use since it's primarily developed in JavaScript
2. Selenium can test web applications against various browsers like Firefox, Chrome, Opera, and Safari
3. Tests can be coded in several programming languages like Java, Python, Perl, PHP, and Ruby

4. Selenium is platform-independent, meaning it can deploy on Windows, Linux, and Macintosh
5. Selenium can be integrated with tools like JUnit and TestNG for test management.

3.6 SELENIUM SUITE OF TOOLS

Selenium has a dedicated suite that facilitates easy testing of web applications.

Fig: Selenium suite

3.6.1 Selenium Integrated Development Environment (IDE):

Developed by Shinya Kasatani in 2006, Selenium IDE is a browser extension for Firefox or Chrome that automates functionality. Typically, IDE records user interactions on the browser and exports them as a reusable script.

IDE was developed to speed up the creation of automation scripts. It's a rapid prototyping tool and can be used by engineers with no programming knowledge whatsoever.

IDE ceased to exist in August 2017 when Firefox upgraded to a new Firefox 55 version, which no longer supported Selenium IDE. Applitools rewrote the old Selenium IDE and released a new version in 2019. The latest version came with several advancements.

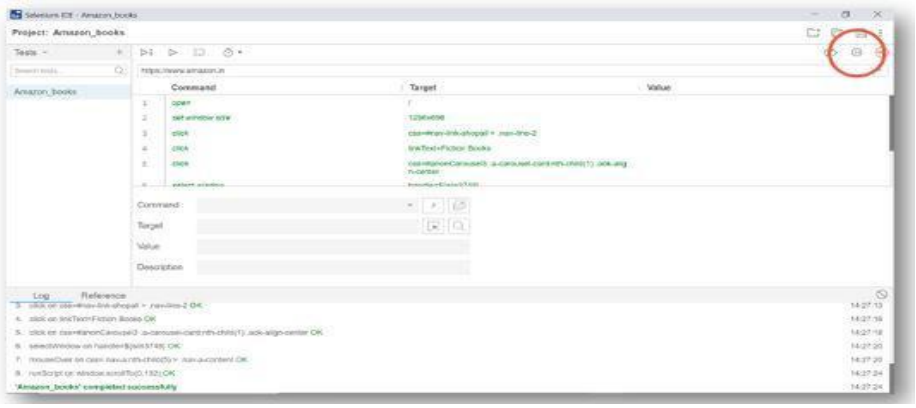


Fig: 3.6.1 Selenium IDE

Selenium IDE has a few shortcomings:

1. It does not support data-driven testing
2. It can't perform database testing
3. It can't provide a detailed test report
4. It can't export to WebDriver scripts

3.6.2 Selenium Remote Control (RC):

Paul Hammant developed Selenium Remote Control (RC). Before we dive into RC, it's important to know why RC came to be in the first place.

Initially, a tool called Selenium-Core was built. It was a set of JavaScript functions that interpreted and executed Selenese commands using the browser's built-in JavaScript interpreter. Selenium-Core was then injected into the web browser.

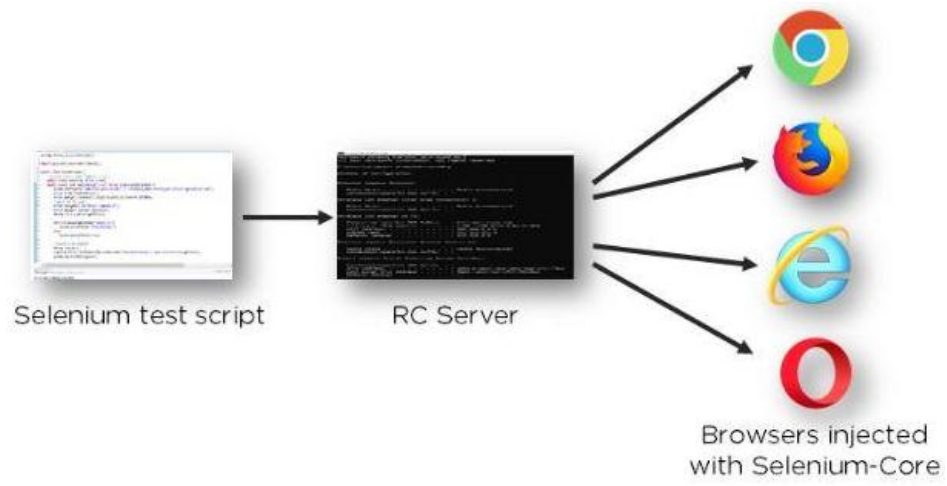
Now, let's consider a JavaScript, test.js used by google.com. This program can access pages like google.com/mail or google.com/login within the google.com domain.

Fig: 3.6.2 Selenium RC

However, the program can't access elements of other domains like yahoo.com. Local copies of Selenium-Core and the web browser had to be installed so they belonged to the same domain. This is called the Same

Origin Policy, and Selenium RC was introduced to address this limitation. The server acts as a client configured HTTP proxy and "tricks" the browser into believing that Selenium Core and the web application being tested come from the same origin.

Hence, Selenium RC is a server written in Java that makes provision for writing application tests in various programming languages like Java, C#, Perl, PHP, Python, etc. The RC server accepts commands from the user program and passes them to the browser as Selenium-Core JavaScript commands.



3.6.3 Selenium WebDriver:

Developed by Simon Stewart in 2006, Selenium WebDriver was the first cross-platform testing framework that could configure and control the browsers on the OS level. It served as a programming interface to create and run test cases.

Unlike Selenium RC, WebDriver doesn't require a core engine like RC and interacts natively with browser applications.

WebDriver also supports various programming languages like Python, Ruby, PHP, and Perl, among others, and can be integrated with frameworks like TestNG and JUnit for test management.

The architecture of Selenium WebDriver is simple and easy to understand:

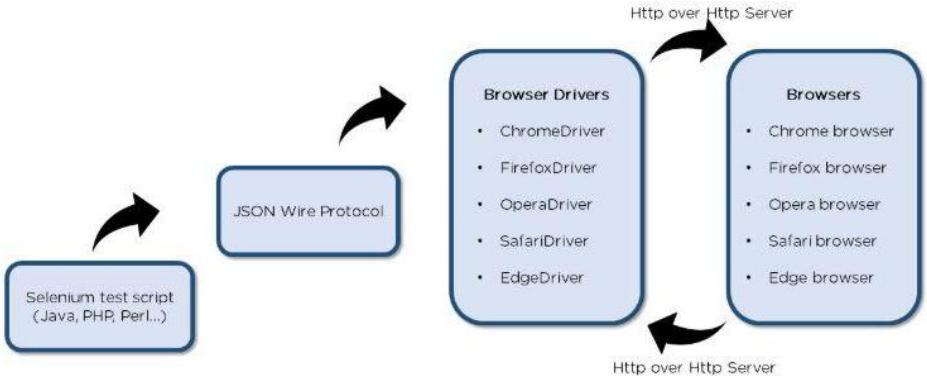


Fig: 3.6.3 Selenium WebDriver architecture

- **Selenium test script:** Selenium test script is the test code written in any programming language be it Java, Perl, PHP, or Python that can be interpreted by the driver.
- **JSON Wire Protocol:** JSON Wire Protocol provides a transport mechanism to transfer data between a server and a client. JSON Wire Protocol serves as an industry standard for various web services.
- **Browser drivers:** Selenium uses drivers specific to each browser to establish a secure connection with the browser.
- **Browsers:** Selenium WebDriver supports various web browsers to test and run applications on.

3.6.4 Selenium Grid:

Patrick Lightbody developed a grid with the primary objective of minimizing the test execution time. This was facilitated by distributing the test commands to different machines simultaneously. Selenium Grid allows the parallel execution of tests on different browsers and different operating systems. Grid is exceptionally flexible and integrates with other suite components for simultaneous execution.

Fig: 3.6.4 Selenium Grid

The Grid consists of a hub connected to several nodes. It receives the test to be executed along with information about the operating system and browser to be run on and picks a node that conforms to the requirements (browser and platform), passing the test to that node. The node now runs the browser and executes the selenium commands within it.

3.7 ADVANTAGES OF SELENIUM TESTING

1. Selenium has proven to be accurate with results thus making it extremely reliable
2. Since selenium is open-source, anybody willing to learn testing can begin at no cost
3. Selenium supports a broad spectrum of programming languages like Python, PHP, Perl, and Ruby
4. Selenium supports various browsers like Chrome, Firefox, and Opera, among others
5. Selenium is easy to implement and doesn't require the engineer to have in-depth knowledge of the tool
6. Selenium has plenty of re-usability and add-ons

As an important aspect of learning what Selenium is, let us understand the limitations of Selenium testing.

3.8 LIMITATIONS OF SELENIUM TESTING

1. Since Selenium is open-source, it doesn't have a developer community and hence doesn't have a reliable tech support
2. Selenium cannot test mobile or desktop applications
3. Selenium offers limited support for image testing
4. Selenium has limited support for test management. Selenium is often integrated with tools like JUnit and TestNG for this purpose
5. You may need knowledge of programming languages to use Selenium

3.9 WHAT IS SELENIUM IDE?

Shinya Kasatani developed Selenium Integrated Development Environment (IDE) in 2006 as a Firefox plugin that helps create tests. IDE is an easy-to-use interface that records user interactions on the browser and exports them as a reusable script.

Selenium IDE is part of the Selenium suite and was developed to speed up the creation of automation scripts. It's a rapid prototyping tool and can be used by engineers with no programming knowledge whatsoever.

3.10 ADVANCEMENTS WITH NEW SELENIUM IDE

In 2017, Firefox upgraded to a new Firefox 55 version, which no longer supported Selenium IDE. Since then, the original version of Selenium IDE ceased to exist. However, Applitools rewrote the old Selenium IDE and released a new version recently.

This new version comes with several new advancements:

- Support for both Chrome and Firefox
- Improved locator functionality
- Parallel execution of tests using Selenium command line runner
- Provision for control flow statements
- Automatically waits for the page to load
- Supports embedded JavaScript code-runs
- IDE has a debugger which allows step execution, adding breakpoints
- Support for code exports

3.11 WORKING PRINCIPLE OF SELENIUM IDE

IDE works in three stages: recording, playing back and saving.

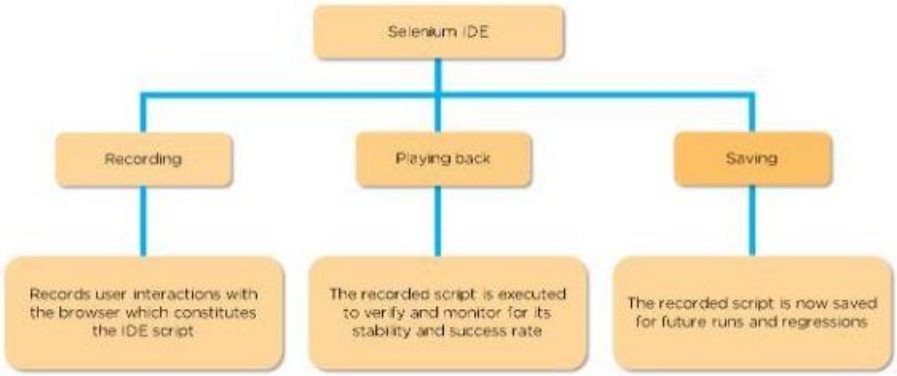


Fig: 3.11 Selenium IDE working principle

In the next section, we'll learn about the three stages in detail, but before we begin, let's acquaint ourselves with the installation of IDE.

1) Recording:

IDE allows the user to record all of the actions performed in the browser. These recorded actions as a whole are the test script.

2) Playing Back:

The recorded script can now be executed to ensure that the browser is working accordingly. Now, the user can monitor the stability and success rate.

3) Saving:

The recorded script is saved with a ".side" extension for future runs and regressions.

3.12 SELENIUM LATEST VERSION

1. What is Selenium 4.0?

Selenium 4 is the latest version of selenium; Simon Stewart founder of selenium has announced Selenium 4 at the Selenium Conference in Bangalore which consists of few major updates and its planned to release after October 2019

2. Features of Selenium 4.0?

2.1 WebDriver became W3C (World Wide Web Consortium) standardization:

The change with Selenium 4 is the standardizing the WebDriver as per the W3C standards. W3C standards encourage compatibility across different software implementations of the WebDriver API and make the Framework much more stable & reduce compatibility issues across different Web Browsers.

A test in Selenium 3 communicates with the browser at the end node through the JSON wire protocol. Testing tools such as Appium and iOS Driver which are related to mobile testing heavily rely on JSON wire protocol.

With Selenium 4 adapting W3C Protocol for communication between the driver and browser, Tests will now be able to directly communicate the browser without using API encoding/decoding. Which would help in mobile testing users

2.2 Improved Selenium Grid:

The Selenium Grid code has been modified with many changes and needed improvements; the console of the selenium grid has been restructured. It allows you to execute test cases in parallel on multiple browsers and systems as well as operating systems.

The UI for Selenium Grid is now with a more user-friendly UI that would have all the relevant information about the sessions running, capacity, etc.

And also, Basic support for utilizing Docker Containers with new Grid Server is been added.

2.3 Support for browsers:

Native support for Opera and PhantomJS going forward will be removed. Whereas users who want to test Opera can opt Chrome since Opera is based on Chromium and for PhantomJS users can use Chrome or Firefox in headless mode. And Selenium Server now no longer includes HtmlUnit by default.

2.4 New Selenium 4 IDE (Chrome & Firefox):

Selenium IDE is a record and playback tool is now available with more advanced capabilities and features.

New plug-in: The new version plugin will allow you to run Selenium on any browser vendor and can declare our own locator strategy.

New CLI runner: The new CLI runner is completely based on node.js instead of old HTML-based runner and will have capabilities like WebDriver Playback and Parallel Execution to support for parallel execution and provide reports like passed and failed tests. The new Selenium IDE runner is completely based on WebDriver.

2.5 Detailed Documentation:

Selenium users face difficulties as there is no proper updated documentation of SelniumNow, SeleniumHQ promises to deliver us update documentation along with the 4.0 version.

2.6 Better Analysis:

Logging and debugging details will be improved to accelerate the resolution of script issues for testers.

3. Impacts of Selenium 4.0?

In addition to these new capabilities, arriving of Selenium 4 may require few changes in your existing selenium 2.X/3.X tests, Let's look at those areas of your tests that require to be updated with Selenium 4

- The getPosition & getSize methods were replaced by getRect method and the setPosition & setSize methods were replaced by setRect method.
- Now you can configure the location of your Safari driver using the "webdriver.safari.driver" system property.
- Element screenshotting is possible in Selenium 4.
- Fullscreen and minimize methods have been added, so that the driver window can now be full screen or minimized and all window manipulation commands are now supported.
- **Added driver.switchTo().parentFrame():** We can use it to go from the child frame to the parent frame directly.
- **Changes added to Chrome driver:**
  - **sendDevToolsCommand():** The sendDevToolsCommand() method sends an arbitrary dev tools command to the browser and returns a promise that will be resolved when the command has finished.
  - **setDownloadPath():** The setDownloadPath() method sends a DevTools command to change Chrome's download directory and

returns a promise that will be resolved when the command has finished.

- **Changes to Firefox driver:** Added installAddon(path) method to install a new addon within the current session. This function will return an "id" that can be used to uninstall the addon using uninstallAddon() method.

- Options class now extends Capabilities class for Chrome, Firefox, IE & Safari.
- Changes wrt to Errors such as Added ElementClickInterceptedError, InsecureCertificateError & Removed ElementNotVisibleError.
- Removed the firefox.profile class. All its functionality is now provided directly by firefox.options.

3.13 INSTALLATION OF SELENIUM IDE

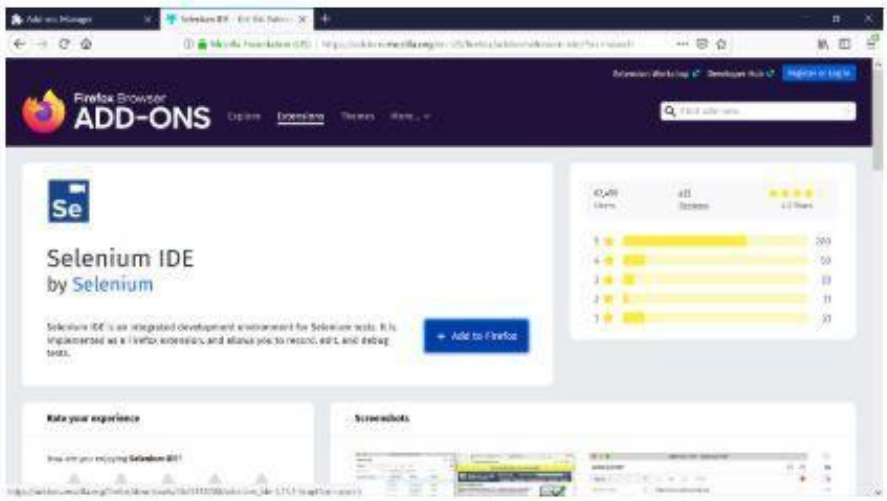
Step 1: Open the Firefox browser.

Step 2: Click on the menu in the top right corner.

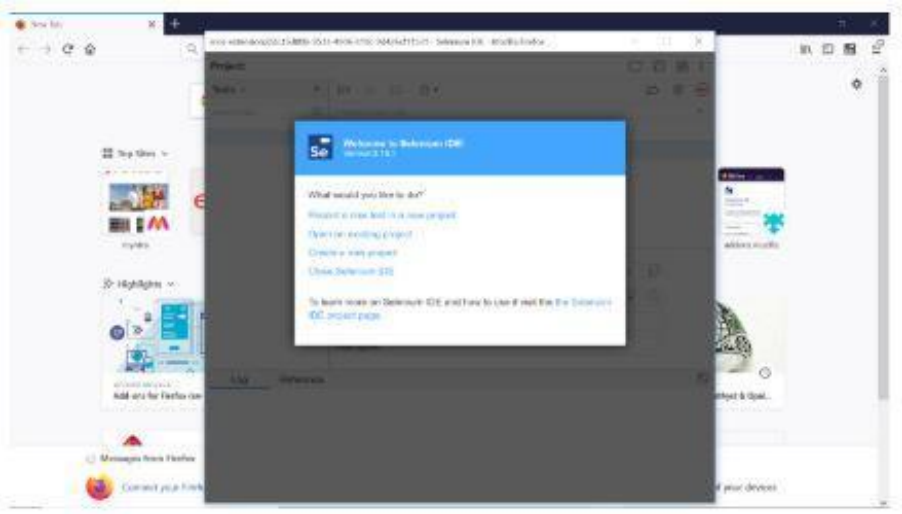
Step 3: Click on Add-ons in the drop-down box.

Step 4: Click on find more add-ons and type "Selenium IDE."

Step 5: Click on Add to Firefox



Once installed, the Selenium IDE icon appears on the top right corner of the browser. When clicked, a Welcome message appears.



Now that we went through the installation,

3.14 DEMO TEST

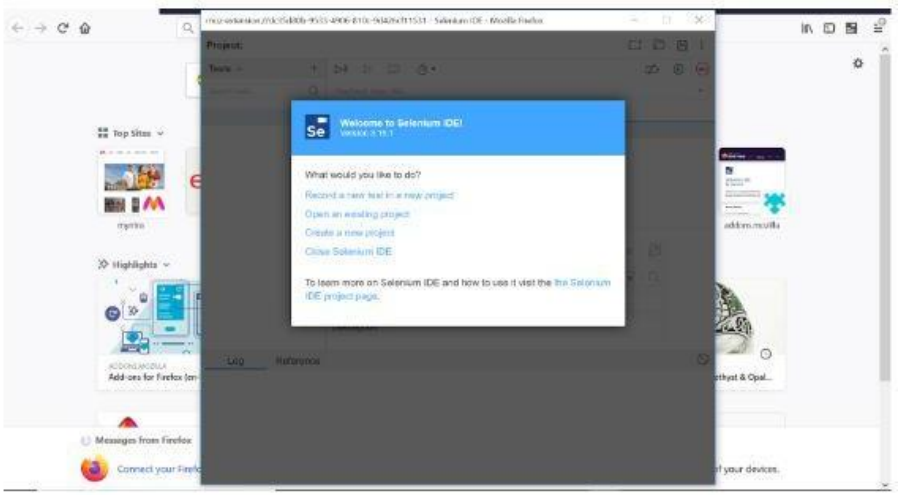
let's create our first test. Consider the following use case for the tutorial:

- Navigate to https://www.facebook.com
- Provide a dummy userID and password
- Log in with these credentials
- Assert title of the application

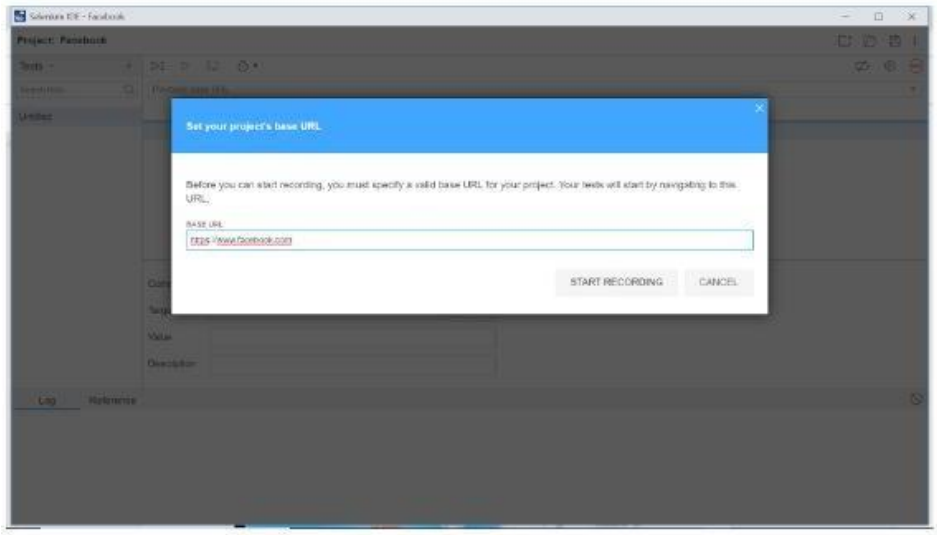
A Simple Demo:

Step 1: Launch your Firefox menu and open the Selenium IDE plugin.

Step 2: Select "Record a test in a new project." Provide the name for your test.



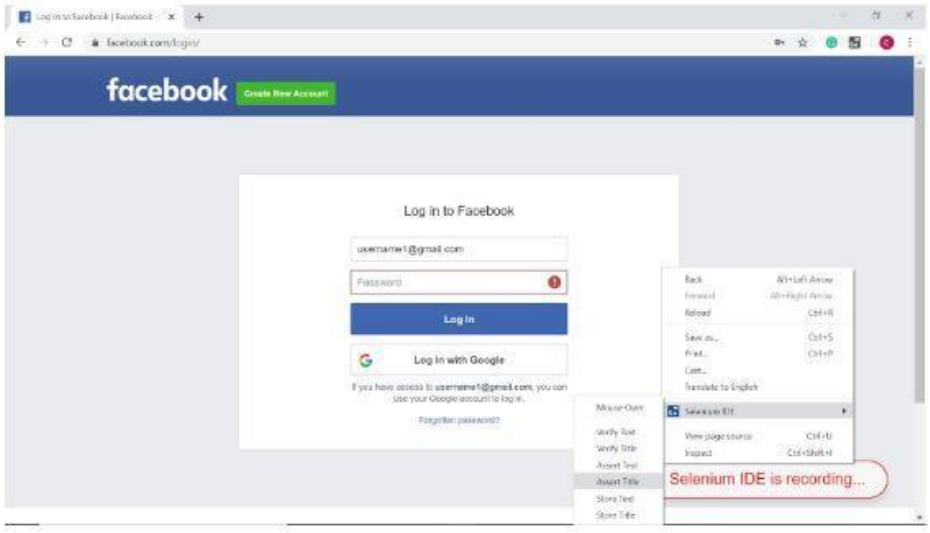
Step 3: Provide a link to the Facebook webpage. The IDE starts recording by navigating to this web page. To continue, click on, OK.



Step 4: Once the website opens, type "username1" in the username field and a dummy password for the password field.

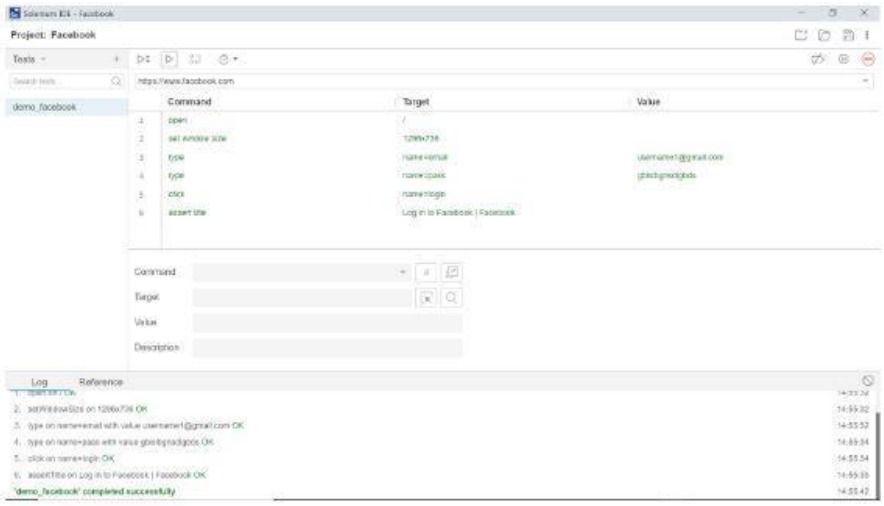
Step 5: Click on "Log In."

Step 6: Now, we verify the title of our application. To do that, Right click>>Selenium IDE>>Assert title. As soon as this is done, a test step would be appended in the IDE editor.



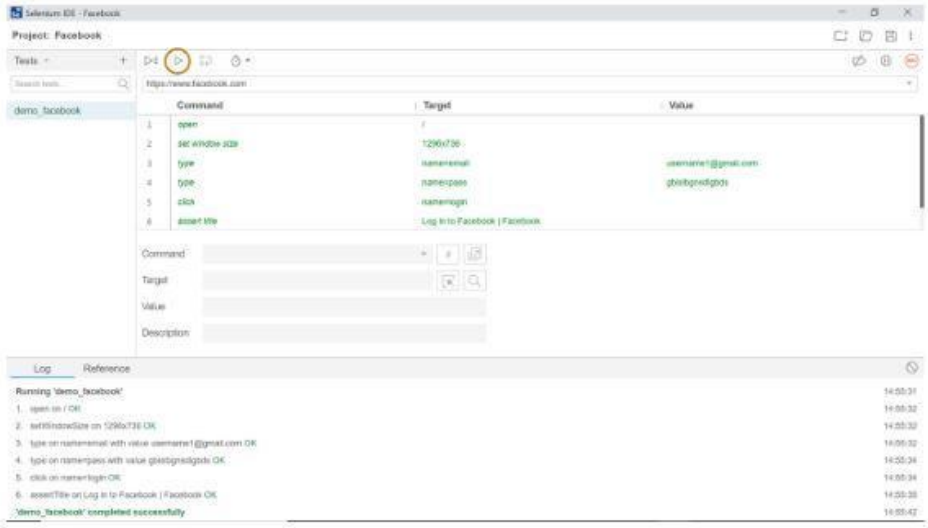
Now you can go back to the IDE editor and click on the Stop icon on the top right corner. With this, we've successfully recorded our test case.

Once the recording is stopped, the editor will look something like this:



1) Playing Back:

Now that the recording is done, we can play it back to verify if the script executes correctly and the browser behaves accordingly. To do this, we can click on the play icon on the menu bar.



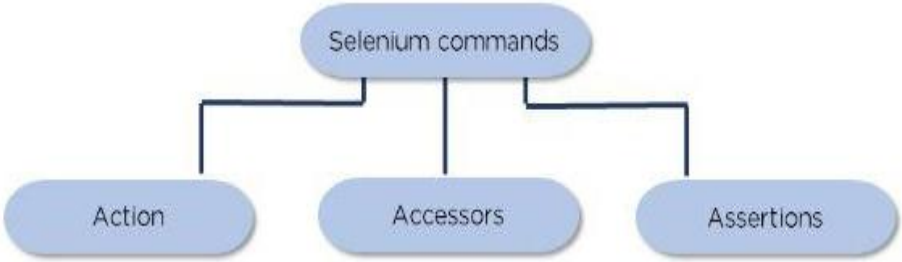
The commands successfully executed are color-coded in green, and the log at the bottom indicates any errors that occurred during the execution. If the script runs successfully, it indicates this by displaying a message.

2) Saving:

Once the test script is successfully executed, you can then save it.

Once clicked, you can browse the location and save it in an appropriate folder. The project is stored with a ".side" extension — this can be used for future runs and regressions.

3) Selenium Commands:



Selenium commands are broadly classified into:

1. **Actions:** Commands that interact directly with the application:

   clickAndWait(),

   typeAndWait()

2. **Enable:** the user to store certain values to a user-defined variable:

4) storeTitle():

Assertions: Verifies the current state of the application along with an expected state. There are different types of assertions:

1. Assert command makes sure that the test execution is terminated in case of failure
2. Verify command ensures script execution even if the verification has failed
3. WaitFor command waits for a specific condition to be met before executing the next test step

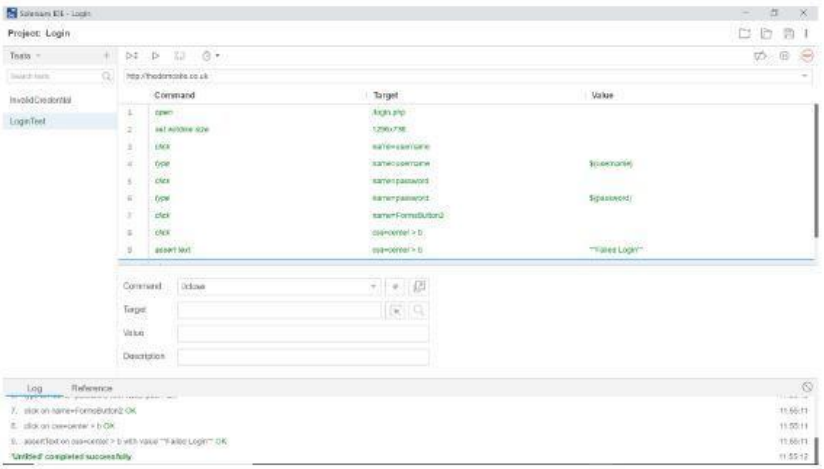
3.15 ADVANCEMENTS WITH THE NEW IDE

1) Re-usability of Test Scripts:

Many of the test scripts in the original version required signing into an application, creating an account, or signing out of an app. As you may see, this is redundant and a waste of time to recreate these test steps over and over. Thankfully, the new Selenium IDE allows one script to run another.

In this test case, let's navigate to http://thedemosite.co.uk/login.php and pass values to the username and password fields. Once the "Failed Login" message appears, we assert the text.

The script in the IDE will look something like this:



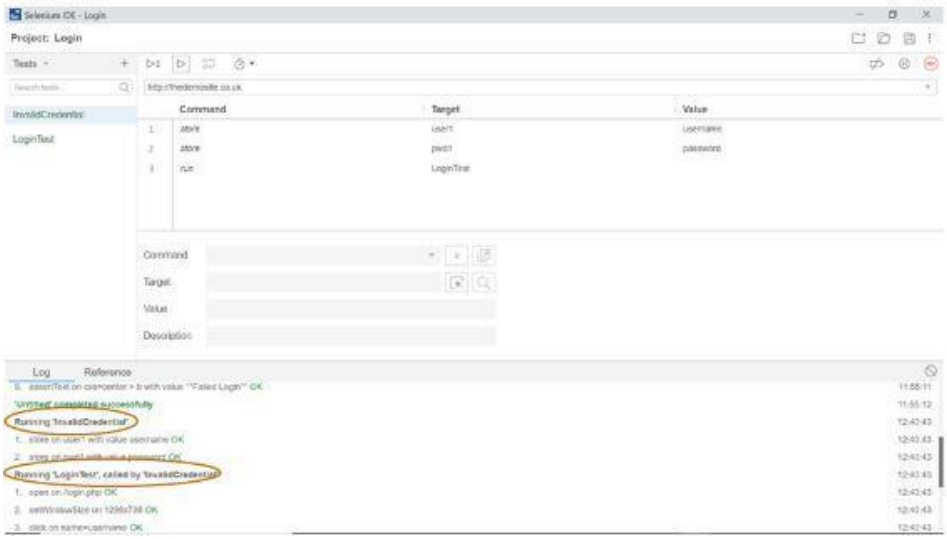
This login function is tested multiple times for different usernames and passwords. To do this, we create a UserID and password.

Click on the commands and add a variable to it rather than hardcoded values.

Now, create another test case and call it “InvalidCredential.” In this test case, we assign values to the username and password variables. Once assigned, we call the original test “LoginTest.”

Use the store command and pass the values “user1” and “pwd1” for the variables username and password, respectively. The Run command runs the target test case (i.e., LoginTest).

Now run the current test case.



As shown above, “LoginTest” is run via “InvalidCredentials.” In this way, the Selenium IDE enables the reusability of test cases, which is similar to passing arguments to a function from the primary function in any programming language.

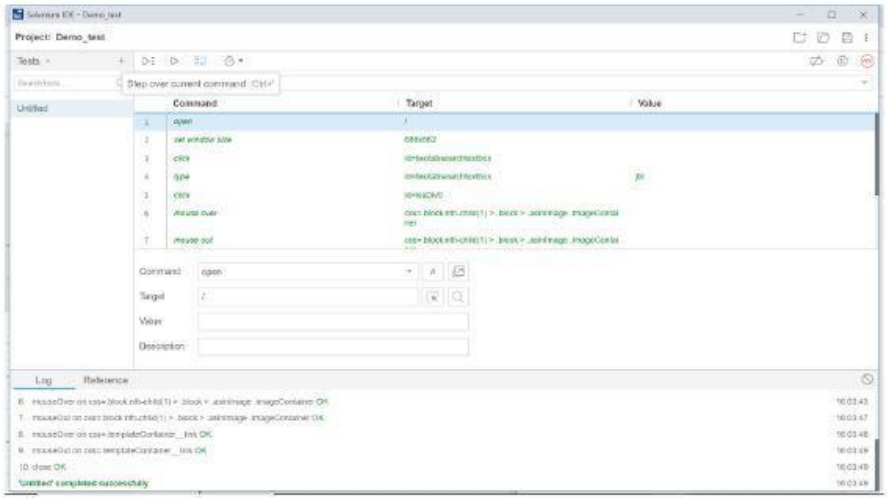
5) Debugging in IDE:

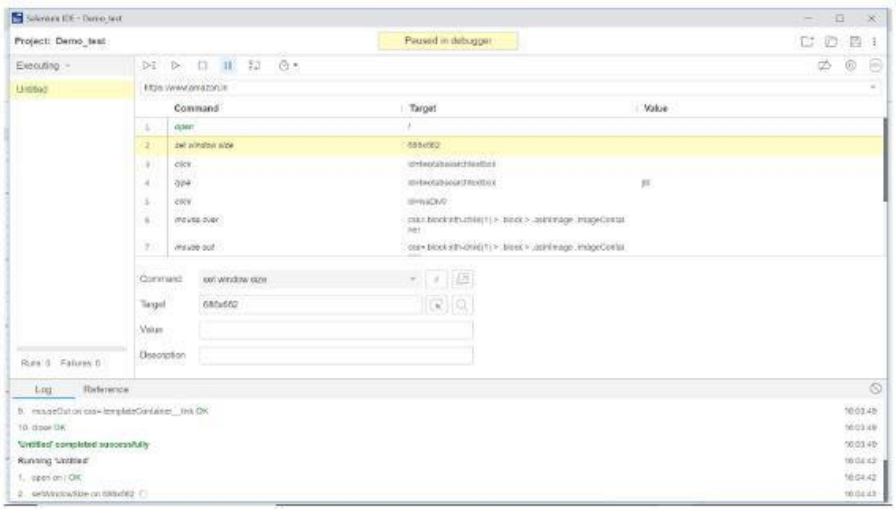
The Selenium IDE facilitates the debugging of test scripts. This enables the user to provide breakpoints where the execution of the test script halts, thus letting the user inspect the browser’s state.

Debugging in IDE is facilitated with the help of the following steps:

1. Step execution: From the test script, you can run every step in succession, controlling the execution of each step. To do this, follow the steps below.

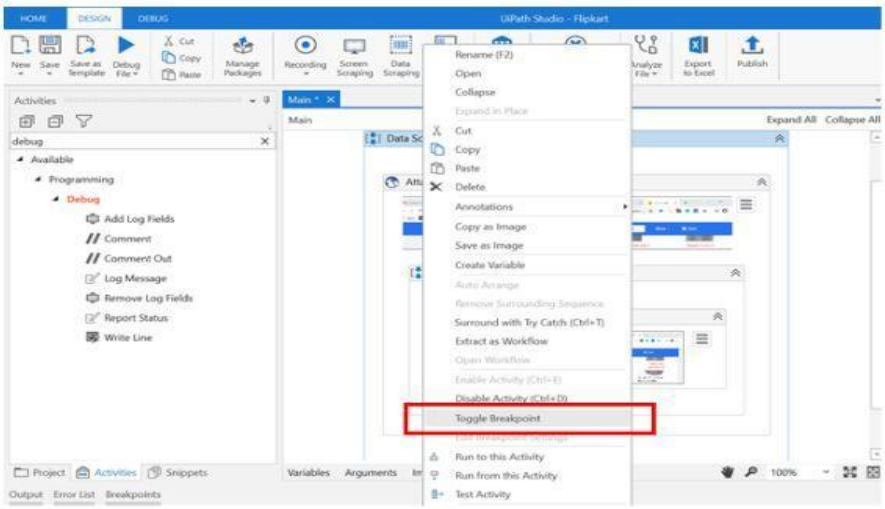
1. Select the command that you would wish to execute.
2. Click on the “Step over current command.”
3. The command executes and pauses before proceeding to the next one.



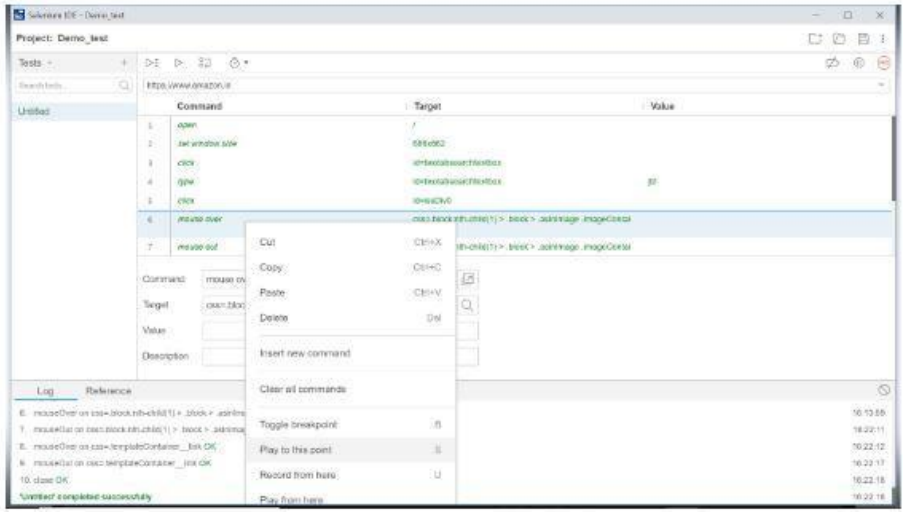


Step execution is useful when the test cases are written are long. You could start checking for errors at any step. This will reveal where the test might be failing.

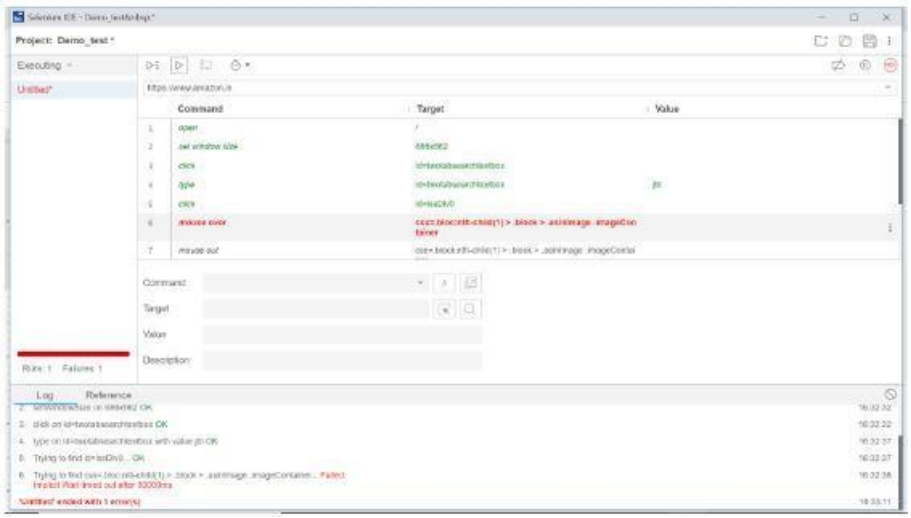
2. Breakpoints: To add breakpoints, click on the digit corresponding to the command you'd like to apply a breakpoint on. Once run, the debugger will execute all the steps before the breakpoint stops the execution.



3. Enter the debugger by using the "Play to this point" option: Another easy way to debug is by using the option as mentioned above. Right-click on the command you'd wish to stop at>>Click on "Play to this point."



4. Pause on Exception: This is a brilliant feature that automatically enters the debug mode in case of an exception. Consider a scenario where the target of a command has been changed.



The target value here is missing a letter causing an error. The script fails, trying to find an incorrect ID.

3.16 SELENIUM WEBDRIVER FIRST SCRIPT

Example 1- Step-by-step instructions for constructing a Selenium script:

Once you have Selenium installed and Drivers installed, you're ready to write Selenium code.

Following are the Eight Basic Components:

Everything Selenium does is send the browser commands to do something or send requests for information. Most of what you'll do with Selenium is a combination of these basic commands:

1. Start the session:

For more details on starting a session read our documentation on opening and closing a browser

```
WebDriver driver = new ChromeDriver();
```

2. Take action on browser:

In this example we are navigating to a web page.

```
driver.get("https://selenium.dev");
```

3. Request browser information:

There are a bunch of types of information about the browser you can request, including window handles, browser size / position, cookies, alerts, etc.

```
driver.getTitle(); // => "Google"
```

4. Establish Waiting Strategy:

Synchronizing the code with the current state of the browser is one of the biggest challenges with Selenium, and doing it well is an advanced topic.

Essentially you want to make sure that the element is on the page before you attempt to locate it and the element is in an interactable state before you attempt to interact with it.

An implicit wait is rarely the best solution, but it's the easiest to demonstrate here, so we'll use it as a placeholder.

```
driver.manage().timeouts().implicitlyWait(Duration.ofMillis(500));
```

5. Find an element:

The majority of commands in most Selenium sessions are element related, and you can't interact with one without first finding an element

```
WebElement searchBox = driver.findElement(By.name("q"));
```

```
WebElement searchButton = driver.findElement(By.name("btnK"));
```

6. Take action on element:

There are only a handful of actions to take on an element, but you will use them frequently.

```
searchBox.sendKeys("Selenium");
```

```
searchButton.click();
```

7. Request element information:

Elements store a lot of information that can be requested. Notice that we need to relocate the search box because the DOM has changed since we first located it.

```
driver.findElement(By.name("q")).getAttribute("value"); // => "Selenium"
```

8. End the session:

This ends the driver process, which by default closes the browser as well. No more commands can be sent to this driver instance.

```
driver.quit();
```

combine these 8 things into a complete script:

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class HelloSelenium {
  public static void main(String[] args) {
    driver = new ChromeDriver();

    driver.get("https://google.com");

    driver.getTitle(); // => "Google"

    driver.manage().timeouts().implicitlyWait(Duration.ofMillis(500));

    WebElement searchBox = driver.findElement(By.name("q"));
    WebElement searchButton = driver.findElement(By.name("btnK"));

    searchBox.sendKeys("Selenium");
    searchButton.click();

    searchBox = driver.findElement(By.name("q"));
    searchBox.getAttribute("value"); // => "Selenium"
```

```
        driver.quit();
    }
}
```

Example 2- To create a WebDriver script that would:

1. fetch Mercury Tours' homepage
2. verify its title
3. print out the result of the comparison
4. close it before ending the entire program.

WebDriver Code:

Below is the actual WebDriver code for the logic presented by the scenario above

```
package newproject;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
//comment the above line and uncomment below line to use Chrome
//import org.openqa.selenium.chrome.ChromeDriver;
public class Sample {
   public static void main(String[] args) {
      // declaration and instantiation of objects/variables

       System.setProperty("webdriver.gecko.driver","C:\\geckodriver.exe")
;
            WebDriver driver = new FirefoxDriver();
            //comment the above 2 lines and uncomment below 2 lines to
use Chrome

       //System.setProperty("webdriver.chrome.driver","G:\\chromedriver.
exe");
            //WebDriver driver = new ChromeDriver();

      String baseUrl = "http://demo.guru99.com/test/newtours/";
      String expectedTitle = "Welcome: Mercury Tours";
      String actualTitle = "";

      // launch Fire fox and direct it to the Base URL
```

```
        driver.get(baseUrl);
        // get the actual value of the title
        actualTitle = driver.getTitle();
        /*
         * compare the actual title of the page with the expected one and print
         * the result as "Passed" or "Failed"
         */
        if (actualTitle.contentEquals(expectedTitle)){
            System.out.println("Test Passed!");
        } else {
            System.out.println("Test Failed");
        }
        //close Fire fox
        driver.close();
    }
}
```

Explaining the code:

Importing Packages

To get started, you need to import following two packages:

1. **org.openqa.selenium:** contains the WebDriver class needed to instantiate a new browser loaded with a specific driver
2. **org.openqa.selenium.firefox.FirefoxDriver:** contains the FirefoxDriver class needed to instantiate a Firefox-specific driver onto the browser instantiated by the WebDriver class

If your test needs more complicated actions such as accessing another class, taking browser screenshots, or manipulating external files, definitely you will need to import more packages.

Instantiating objects and variables

Normally, this is how a driver object is instantiated.

```
WebDriver driver = new FirefoxDriver();
```

A FirefoxDriver class with no parameters means that the default Firefox profile will be launched by our Java program. The default Firefox profile is similar to launching Firefox in safe mode (no extensions are loaded).

For convenience, we saved the Base URL and the expected title as variables.

1) Launching a Browser Session:

WebDriver's **get()** method is used to launch a new browser session and directs it to the URL that you specify as its parameter.

```
driver.get(baseUrl);
```

2) Get the Actual Page Title:

The WebDriver class has the **getTitle()** method that is always used to obtain the page title of the currently loaded page.

```
actualTitle = driver.getTitle();
```

3) Compare the Expected and Actual Values:

This portion of the code simply uses a basic Java if-else structure to compare the actual title with the expected one.

```
if (actualTitle.contentEquals(expectedTitle)) {
    System.out.println("Test Passed!");
} else {
    System.out.println("Test Failed!");
}
```

4) Terminating a Browser Session:

The "**close()**" method is used to close the browser window.

```
driver.close();
```

Example-3 Selenium WebDriver- First Test Case(Automation Testing):

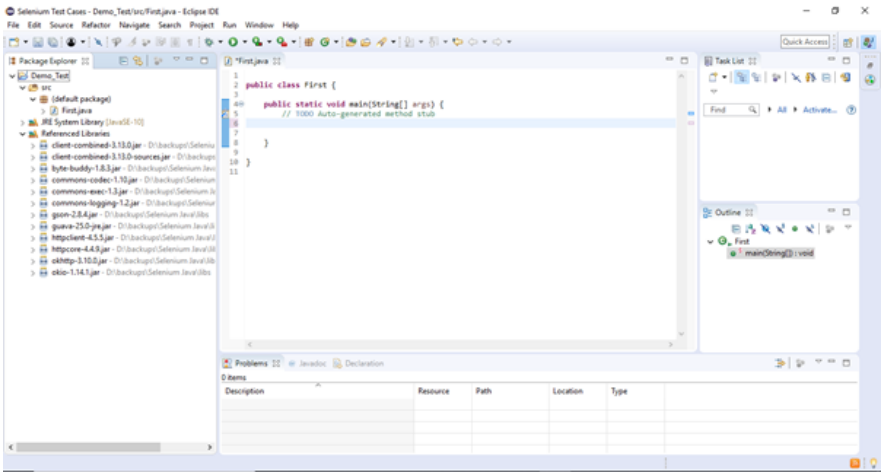
To create your First Selenium Automation Test Script.

Under this test, we will automate the following scenarios:

- Invoke Google Chrome browser.
- Open URL: www.google.com
- Click on the Google Search text box.
- Type the value "javatpoint tutorials"
- Click on the Search button.

We will create our test case step by step to give you a complete understanding of each component in detail.

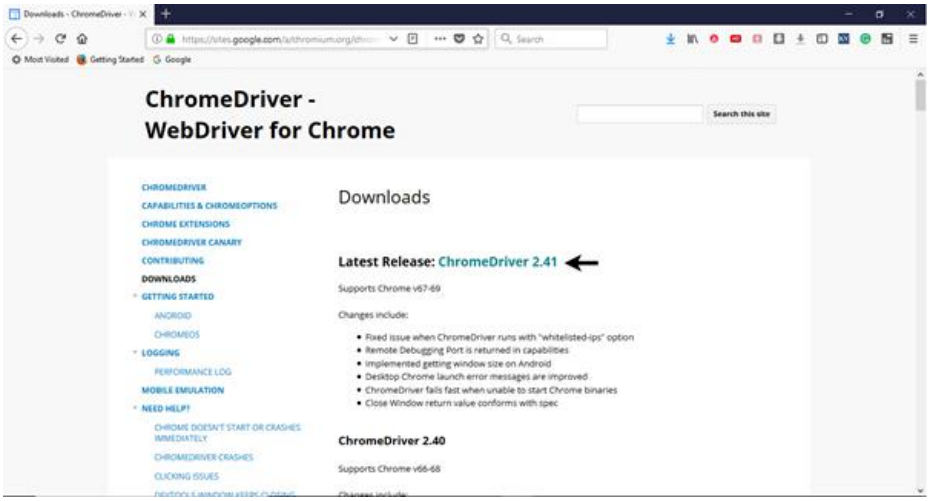
Step1: Launch Eclipse IDE and open project "Demo_Test" which we have created in the previous section (Configure Selenium WebDriver) of this Tutorial. We will write our first Selenium test script in the "First.class" file under the "Demo_Test" test suite.



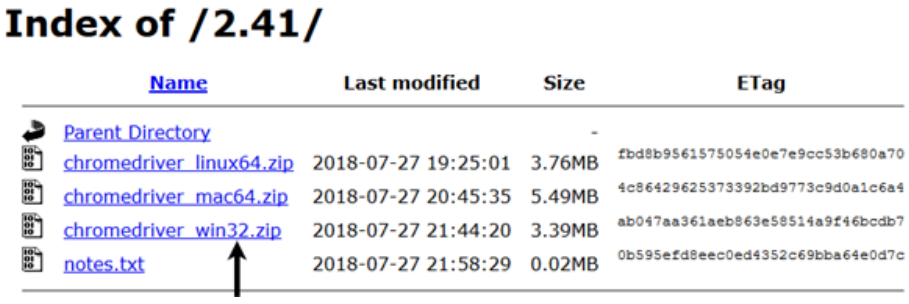
Note: To invoke a browser in Selenium, we have to download an executable file specific to that browser. For example, Chrome browser implements the WebDriver protocol using an executable called ChromeDriver.exe. These executable files start a server on your system which in turn is responsible for running your test scripts in Selenium.

Step2: Open URL https://sites.google.com/a/chromium.org/chromedriver/downloads in your browser.

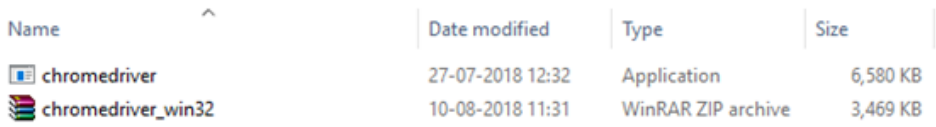
Step3: Click on the "ChromeDriver 2.41" link. It will redirect you to the directory of ChromeDriver executables files. Download as per the operating system you are currently on.



For windows, click on the "chromedriver_win32.zip" download.



The downloaded file would be in zipped format. Unpack the contents in a convenient directory.



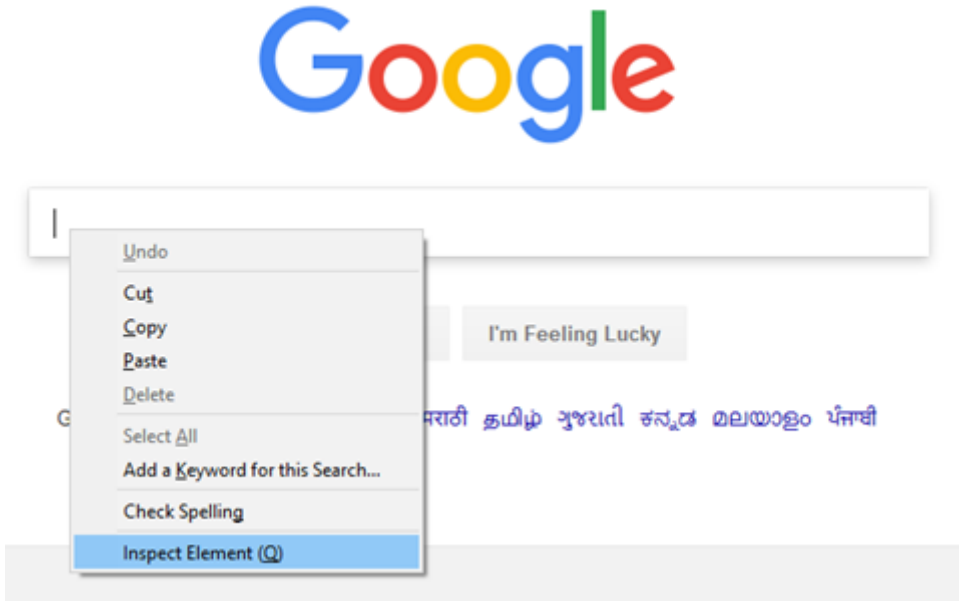
Note: Selenium developers have defined properties for each browser that needs the location of the respective executable files to be parsed in order to invoke a browser. For example, the property defined for Chrome browser - webdriver.chrome.driver, needs the path of its executable file - D:\ChromeDriver\chromedriver.exe in order to launch chrome browser.

```
System.setProperty("webdriver.chrome.driver", "D:\\ChromeDriver\\chromedriver.exe");
```

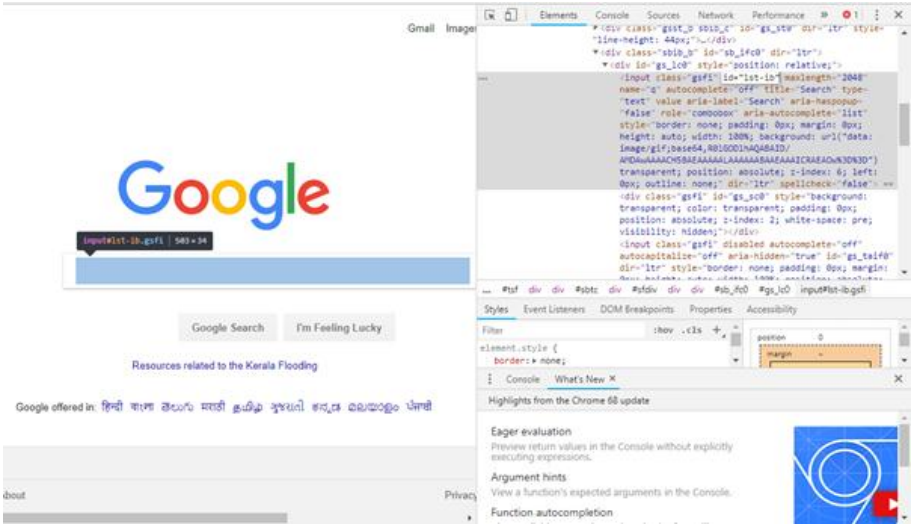
Step4: We would need a unique identification for the web elements like Google Search text box and Search button in order to automate them through our test script. These unique identifications are configured along with some Commands/Syntax to form Locators. Locators help us to locate and identify a particular web element in context of a web application.

The method for finding a unique identification element involves inspection of HTML codes.

- Open URL: https://www.google.com in your Chrome browser.
- Right click on the Google search text box and select Inspect Element.



- It will launch a window containing all the specific codes involved in the development of the test box.



- Pick the value of id element i.e. "lst-ib".

```
<input class="gsfi" id="lst-ib" maxlength="2048"
name="q" autocomplete="off" title="Search" type=
```

- Given below is the Java syntax for locating elements through "id" in Selenium WebDriver.

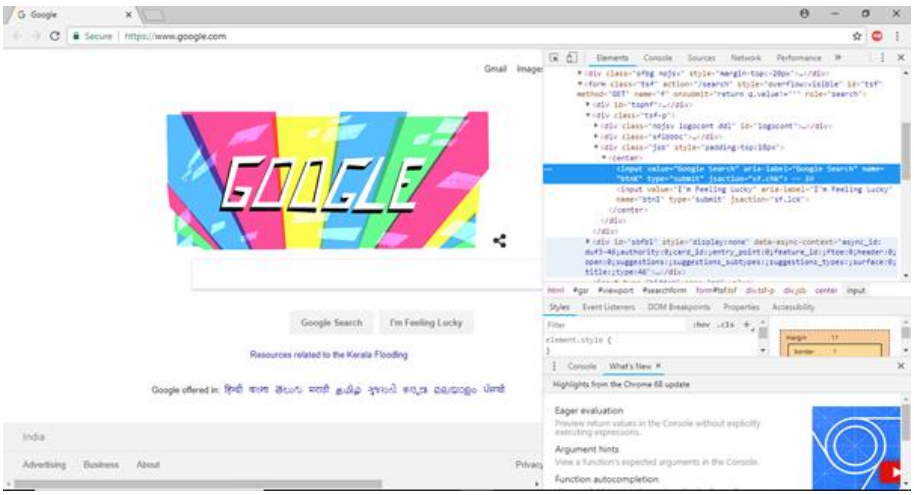
  1. driver.findElement(By.id (<element ID>))

- Here is the complete code for locating Google Search text box in our test script.

  1. driver.findElement(By.id ("lst-ib"))

- Now, right click on the Google Search button and select Inspect Element.

- It will launch a window containing all the specific codes involved in the development of the Google Search button.



- Pick the value of **name** element i.e. "btnK".

```
<input value="Google Search" aria-label="Google Search"
name="btnK" type="submit" jsaction="sf.chk"> == $0
```

- Given below is the Java syntax for locating elements through "name" in Selenium WebDriver.

  1. driver.findElement(By.name (<element name>))

- Here is the complete code for locating Google Search button in our test script.

  1. driver.findElement(By.name ("btnK"))

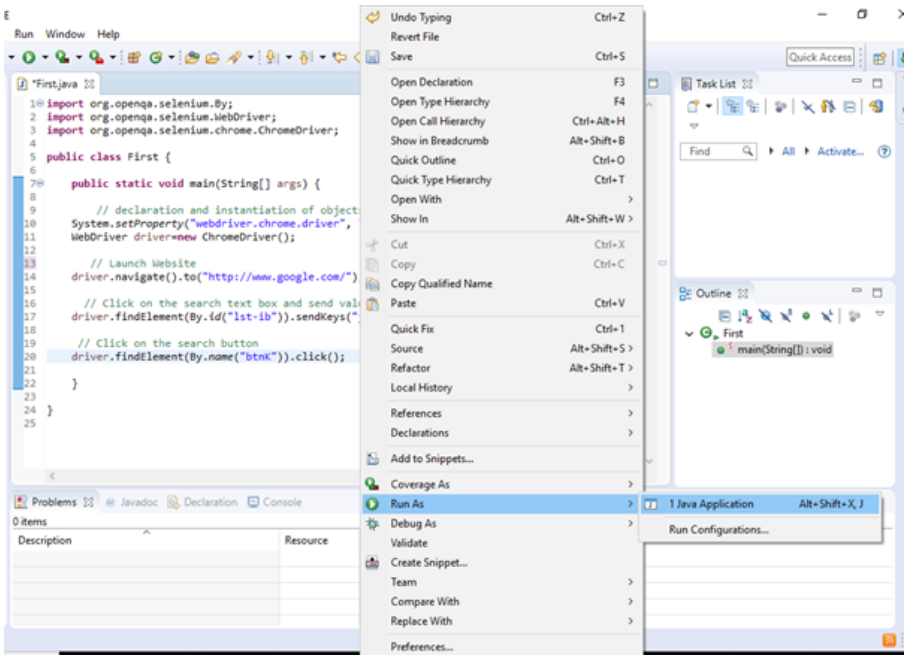
Step5: Now it is time to code. We have embedded comments for each block of code to explain the steps clearly.

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class First {

   public static void main(String[] args) {

   // declaration and instantiation of objects/variables
   System.setProperty("webdriver.chrome.driver",
"D:\\ChromeDriver\\chromedriver.exe");
   WebDriver driver=new ChromeDriver();
```

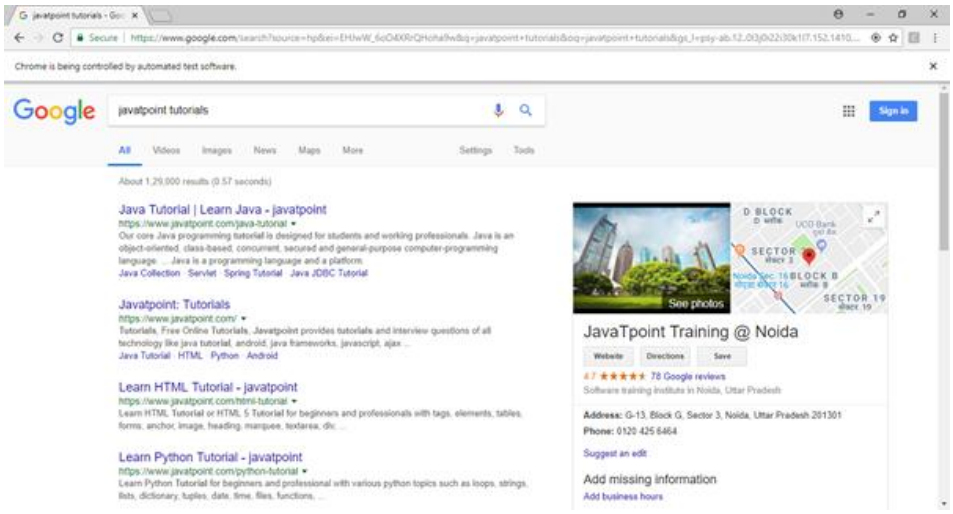
```
// Launch website
    driver.navigate().to("http://www.google.com/");

    // Click on the search text box and send value
    driver.findElement(By.id("lst-ib")).sendKeys("javatpoint tutorials");

    // Click on the search button
    driver.findElement(By.name("btnK")).click();
    }
}
```

The Eclipse code window will look like this:

Step6: Right click on the Eclipse code and select **Run As > Java Application**.



Step7: The output of above test script would be displayed in Google Chrome browser.



- **Explanation of the Code:**

Import Packages/Statements: In java, import statements are used to import the classes present in another packages. In simple words, import keyword is used to import built-in and user-defined packages into your java source file.

1. **org.openqa.selenium.WebDriver**: References the WebDriver interface which is required to instantiate a new web browser.

2. **org.openqa.selenium.chrome.ChromeDriver**: References the ChromeDriver class that is required to instantiate a Chrome-specific driver onto the browser instantiated by the WebDriver class.

- **Instantiating objects and variables:**

A driver object is instantiated through:

WebDriver driver=**new** ChromeDriver();

- **Launch Website:**

To launch a new website, we use navigate().to() method in WebDriver.

driver.navigate().to("http://www.google.com/");

- **Click on an element:**

In WebDriver, user interactions are performed through the use of Locators which we would discuss in later sessions of this tutorial. For now, the following instance of code is used to locate and parse values in a specific web element.

driver.findElement(By.id("lst-ib")).sendKeys("javatpoint tutorials");

3.15 QUESTIONS

Q.1) Define Testing. Explain challenges in manual testing.

Q.2) What is selenium ? Explain the advantages of automation testing.

Q.3) Explain in brief different tools in selenium.

Q.4) Difference between manual testing & automation testing.

Q.5) Explain selenium IDE.

Q.6) Explain steps Installation of Selenium IDE.

Q.7) Record a test case using selenium.

Q.8) Write a simple selenium WebDriver.

3.16 REFERENCES

1) Software Testing Foundations, 4th Edition: A Study Guide for the Certified Tester Exam (Rocky Nook Computing) Fourth Edition,Andreas Spillner, Tilo Linz and Hans Schaefer.

2) Selenium WebDriver, Pearson, Rajeev Gupta, ISBN 9789332526297.

3) Selenium WebDriver Practical Guide - Automated Testing for Web

4) Applications Kindle Edition ,SatyaAvasarala ,ISBN-13: 978-1782168850

5) https://www.guru99.com/selenium-tutorial.html

6) https://www.toolsqa.com/selenium-tutorial/

7) https://www.techlistic.com/p/selenium-tutorials.html

*****

4

INTRODUCTION TO SELENIUM

Unit Structure

4.1 Advancements with the New IDE

4.2 Selenium WebDriver First Script.

4.3 Questions.

4.4 References

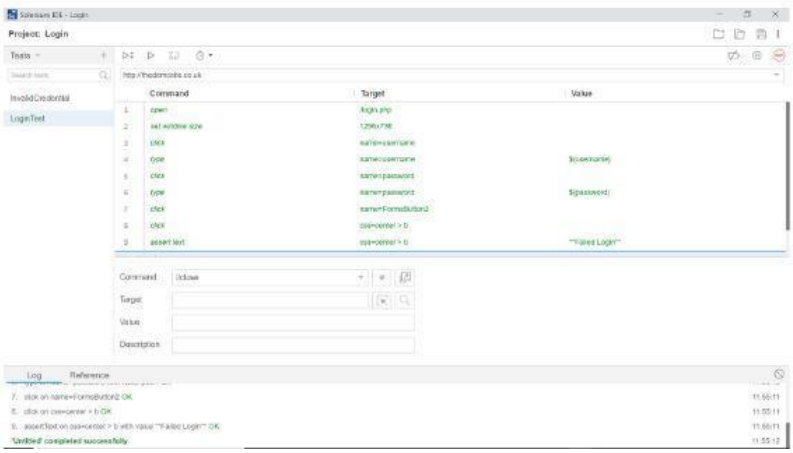
4.1 ADVANCEMENTS WITH THE NEW IDE

1) Re-usability of Test Scripts:

Many of the test scripts in the original version required signing into an application, creating an account, or signing out of an app. As you may see, this is redundant and a waste of time to recreate these test steps over and over. Thankfully, the new Selenium IDE allows one script to run another.

In this test case, let's navigate to http://thedemosite.co.uk/login.php and pass values to the username and password fields. Once the "Failed Login" message appears, we assert the text.

The script in the IDE will look something like this:



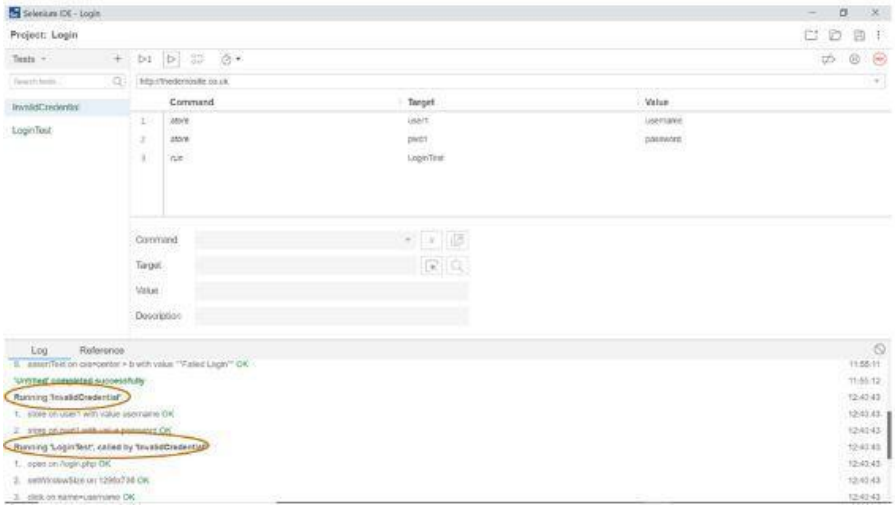
This login function is tested multiple times for different usernames and passwords. To do this, we create a UserID and password.

Click on the commands and add a variable to it rather than hardcoded values.

Now, create another test case and call it “InvalidCredential.” In this test case, we assign values to the username and password variables. Once assigned, we call the original test “LoginTest.”

Use the store command and pass the values “user1” and “pwd1” for the variables username and password, respectively. The Run command runs the target test case (i.e., LoginTest).

Now run the current test case.



As shown above, “LoginTest” is run via “InvalidCredentials.” In this way, the Selenium IDE enables the reusability of test cases, which is similar to passing arguments to a function from the primary function in any programming language.

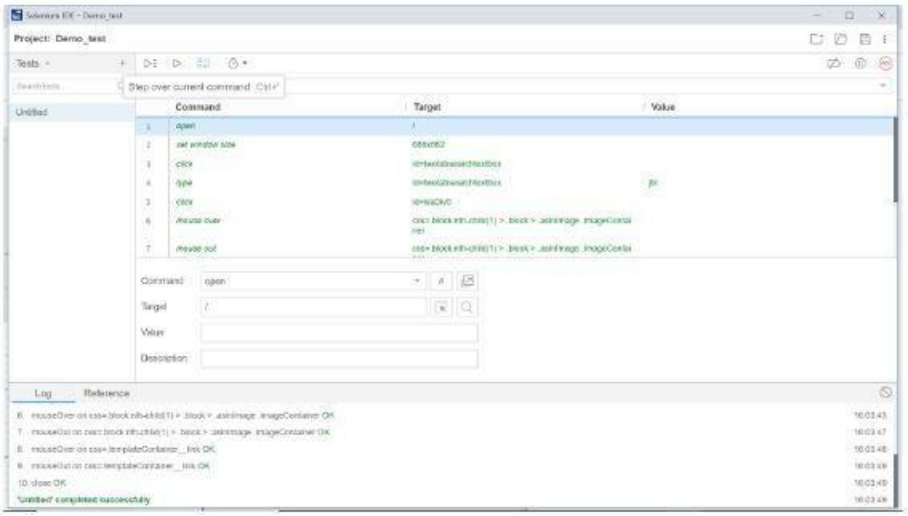
5) Debugging in IDE:

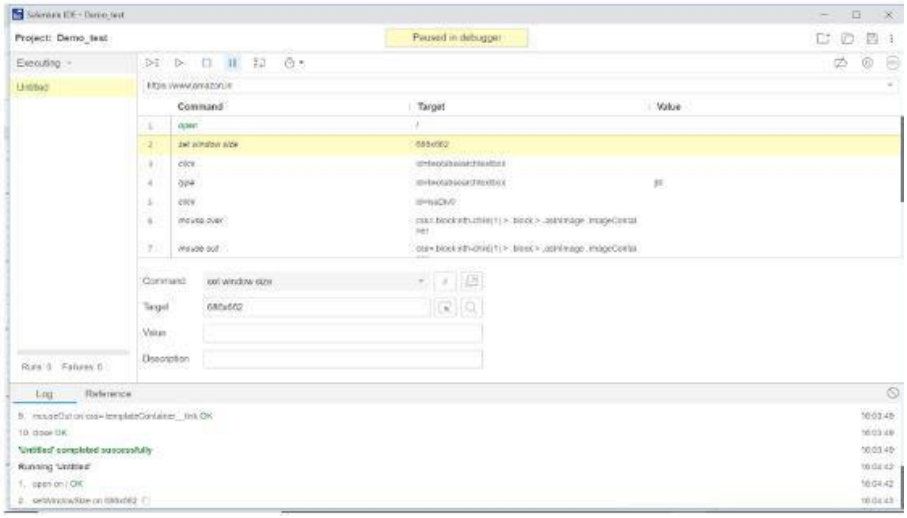
The Selenium IDE facilitates the debugging of test scripts. This enables the user to provide breakpoints where the execution of the test script halts, thus letting the user inspect the browser's state.

Debugging in IDE is facilitated with the help of the following steps:

1. **Step execution:** From the test script, you can run every step in succession, controlling the execution of each step. To do this, follow the steps below.

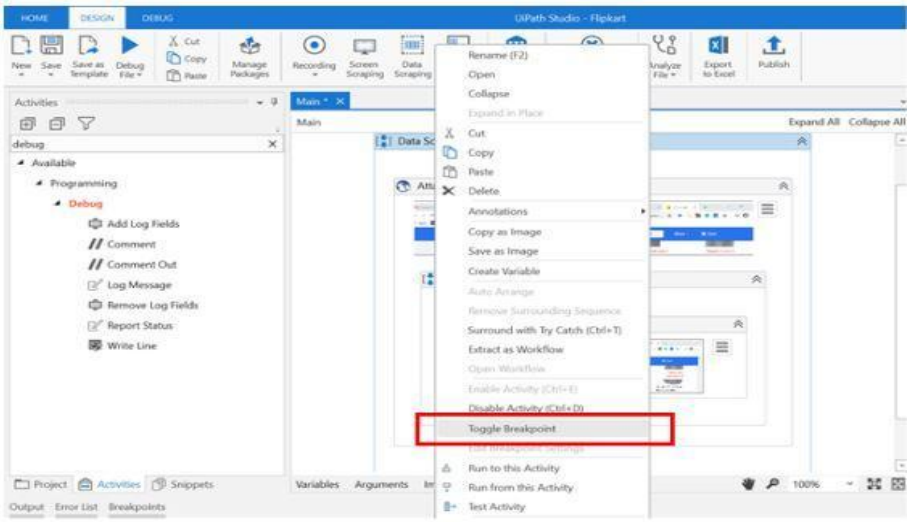
1. Select the command that you would wish to execute.
2. Click on the "Step over current command."
3. The command executes and pauses before proceeding to the next one.



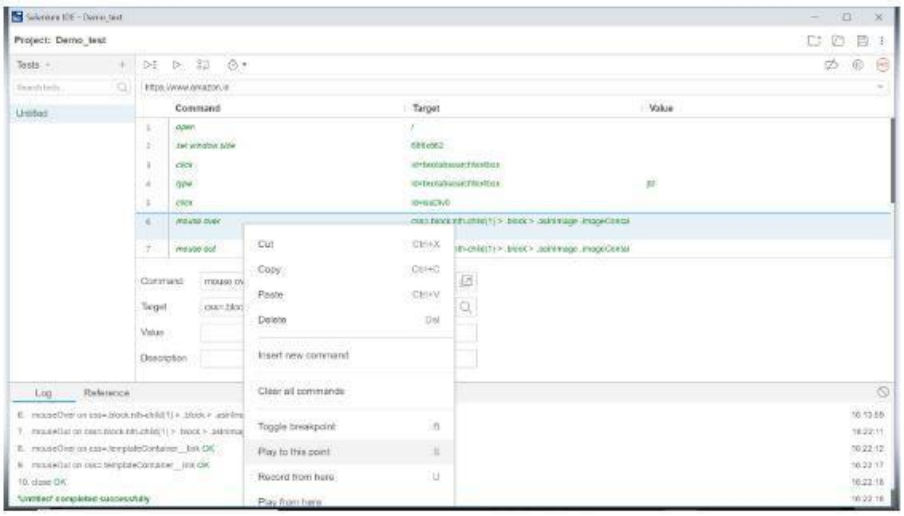


Step execution is useful when the test cases are written are long. You could start checking for errors at any step. This will reveal where the test might be failing.

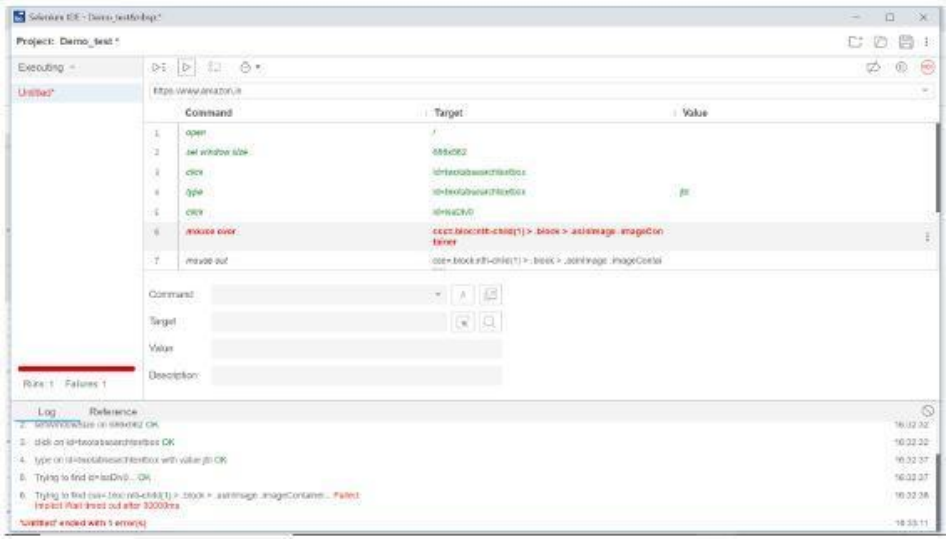
2. **Breakpoints:** To add breakpoints, click on the digit corresponding to the command you'd like to apply a breakpoint on. Once run, the debugger will execute all the steps before the breakpoint stops the execution.



3. **Enter the debugger by using the "Play to this point" option:** Another easy way to debug is by using the option as mentioned above. Right-click on the command you'd wish to stop at>>Click on "Play to this point."



4. **Pause on Exception:** This is a brilliant feature that automatically enters the debug mode in case of an exception. Consider a scenario where the target of a command has been changed.



The target value here is missing a letter causing an error. The script fails, trying to find an incorrect ID.

4.2 SELENIUM WEBDRIVER FIRST SCRIPT

Example 1- Step-by-step instructions for constructing a Selenium script:

Once you have Selenium installed and Drivers installed, you're ready to write Selenium code.

Following are the Eight Basic Components:

Everything Selenium does is send the browser commands to do something or send requests for information. Most of what you'll do with Selenium is a combination of these basic commands:

1. Start the session:

For more details on starting a session read our documentation on opening and closing a browser

```
WebDriver driver = new ChromeDriver();
```

2. Take action on browser:

In this example we are navigating to a web page.

```
driver.get("https://selenium.dev");
```

3. Request browser information:

There are a bunch of types of information about the browser you can request, including window handles, browser size / position, cookies, alerts, etc.

```
driver.getTitle(); // => "Google"
```

4. Establish Waiting Strategy:

Synchronizing the code with the current state of the browser is one of the biggest challenges with Selenium, and doing it well is an advanced topic.

Essentially you want to make sure that the element is on the page before you attempt to locate it and the element is in an interactable state before you attempt to interact with it.

An implicit wait is rarely the best solution, but it's the easiest to demonstrate here, so we'll use it as a placeholder.

```
driver.manage().timeouts().implicitlyWait(Duration.ofMillis(500));
```

5. Find an element:

The majority of commands in most Selenium sessions are element related, and you can't interact with one without first finding an element

```
WebElement searchBox = driver.findElement(By.name("q"));
```

```
WebElement searchButton = driver.findElement(By.name("btnK"));
```

6. Take action on element:

There are only a handful of actions to take on an element, but you will use them frequently.

```
searchBox.sendKeys("Selenium");
```

```
searchButton.click();
```

7. Request element information:

Elements store a lot of information that can be requested. Notice that we need to relocate the search box because the DOM has changed since we first located it.

```
driver.findElement(By.name("q")).getAttribute("value"); // => "Selenium"
```

8. End the session:

This ends the driver process, which by default closes the browser as well. No more commands can be sent to this driver instance.

```
driver.quit();
```

combine these 8 things into a complete script:

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
```

```
public class HelloSelenium {
  public static void main(String[] args) {
      driver = new ChromeDriver();

      driver.get("https://google.com");

      driver.getTitle(); // => "Google"

      driver.manage().timeouts().implicitlyWait(Duration.ofMillis(500));

      WebElement searchBox = driver.findElement(By.name("q"));
      WebElement searchButton = driver.findElement(By.name("btnK"));

      searchBox.sendKeys("Selenium");
      searchButton.click();

      searchBox = driver.findElement(By.name("q"));
      searchBox.getAttribute("value"); // => "Selenium"

      driver.quit();
  }
}
```

Example 2- To create a WebDriver script that would:

1. fetch Mercury Tours' homepage
2. verify its title
3. print out the result of the comparison
4. close it before ending the entire program.

WebDriver Code:

Below is the actual WebDriver code for the logic presented by the scenario above

```
package newproject;

import org.openqa.selenium.WebDriver;
```

```
import org.openqa.selenium.firefox.FirefoxDriver;
//comment the above line and uncomment below line to use Chrome
//import org.openqa.selenium.chrome.ChromeDriver;
public class Sample {
   public static void main(String[] args) {
      // declaration and instantiation of objects/variables

       System.setProperty("webdriver.gecko.driver","C:\\geckodriver.exe")
;
            WebDriver driver = new FirefoxDriver();
            //comment the above 2 lines and uncomment below 2 lines to
use Chrome

       //System.setProperty("webdriver.chrome.driver","G:\\chromedriver.
exe");
            //WebDriver driver = new ChromeDriver();

      String baseUrl = "http://demo.guru99.com/test/newtours/";
      String expectedTitle = "Welcome: Mercury Tours";
      String actualTitle = "";

      // launch Fire fox and direct it to the Base URL
      driver.get(baseUrl);
      // get the actual value of the title
      actualTitle = driver.getTitle();
      /*
       * compare the actual title of the page with the expected one and print
       * the result as "Passed" or "Failed"
       */
      if (actualTitle.contentEquals(expectedTitle)){
         System.out.println("Test Passed!");
```

```
        } else {
            System.out.println("Test Failed");
        }
        //close Fire fox
        driver.close();
    }
}
```

Explaining the code:

Importing Packages

To get started, you need to import following two packages:

1. **org.openqa.selenium:** contains the WebDriver class needed to instantiate a new browser loaded with a specific driver
2. **org.openqa.selenium.firefox.FirefoxDriver**: contains the FirefoxDriver class needed to instantiate a Firefox-specific driver onto the browser instantiated by the WebDriver class

If your test needs more complicated actions such as accessing another class, taking browser screenshots, or manipulating external files, definitely you will need to import more packages.

Instantiating objects and variables

Normally, this is how a driver object is instantiated.

```
WebDriver driver = new FirefoxDriver();
```

A FirefoxDriver class with no parameters means that the default Firefox profile will be launched by our Java program. The default Firefox profile is similar to launching Firefox in safe mode (no extensions are loaded).

For convenience, we saved the Base URL and the expected title as variables.

1) Launching a Browser Session:

WebDriver's **get()** method is used to launch a new browser session and directs it to the URL that you specify as its parameter.

```
driver.get(baseUrl);
```

2) Get the Actual Page Title:

The WebDriver class has the **getTitle()** method that is always used to obtain the page title of the currently loaded page.

```
actualTitle = driver.getTitle();
```

3) Compare the Expected and Actual Values:

This portion of the code simply uses a basic Java if-else structure to compare the actual title with the expected one.

```
if (actualTitle.contentEquals(expectedTitle)) {
    System.out.println("Test Passed!");
} else {
    System.out.println("Test Failed!");
}
```

4) Terminating a Browser Session:

The "**close()**" method is used to close the browser window.

```
driver.close();
```

Example- 3 Selenium WebDriver- First Test Case (Automation Testing):

To create your First Selenium Automation Test Script.

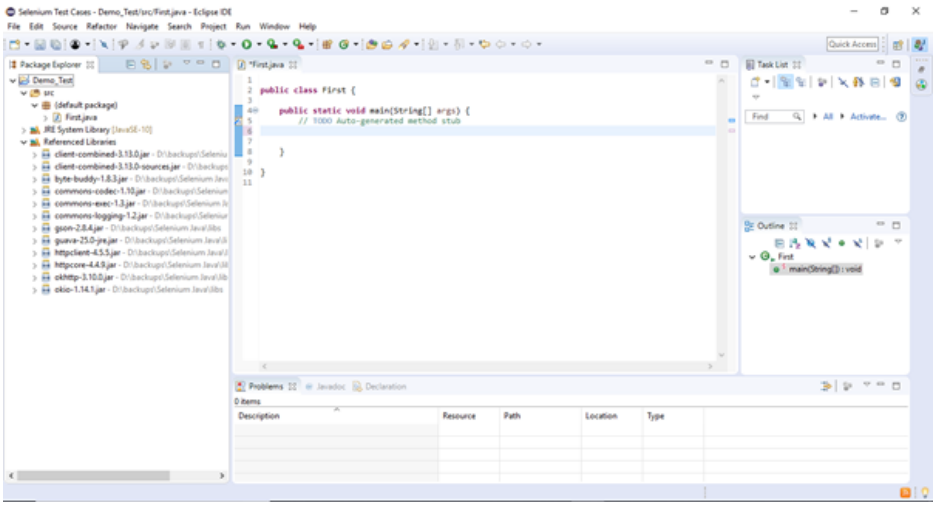
Under this test, we will automate the following scenarios:

- Invoke Google Chrome browser.
- Open URL: www.google.com
- Click on the Google Search text box.
- Type the value "javatpoint tutorials"
- Click on the Search button.

We will create our test case step by step to give you a complete understanding of each component in detail.

Step 1:

Launch Eclipse IDE and open project "Demo_Test" which we have created in the previous section (Configure Selenium WebDriver) of this Tutorial. We will write our first Selenium test script in the "First.class" file under the "Demo_Test" test suite.



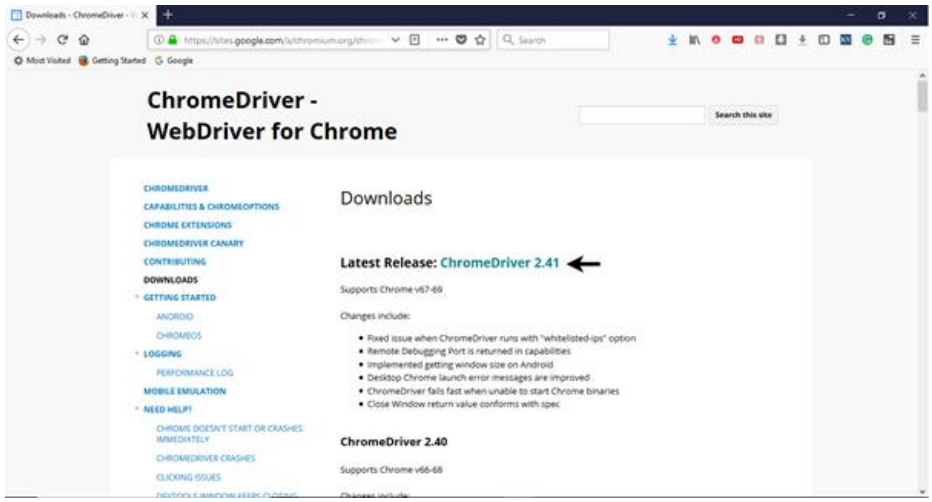
Note: To invoke a browser in Selenium, we have to download an executable file specific to that browser. For example, Chrome browser implements the WebDriver protocol using an executable called ChromeDriver.exe. These executable files start a server on your system which in turn is responsible for running your test scripts in Selenium.

Step 2:

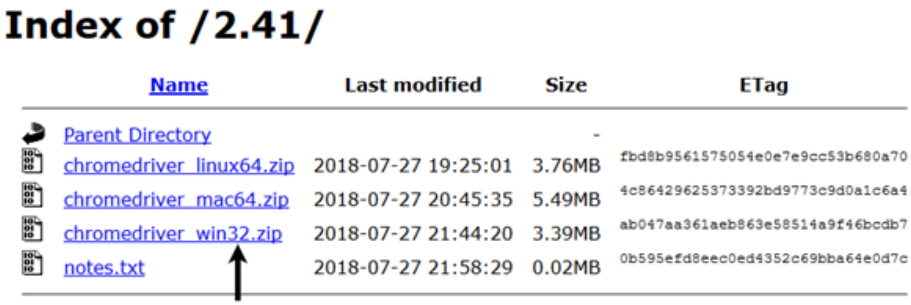
Open URL: https://sites.google.com/a/chromium.org/chromedriver/downloads in your browser.

Step 3:

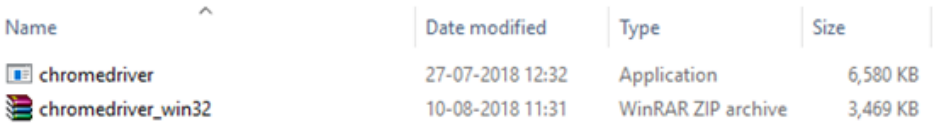
Click on the "ChromeDriver 2.41" link. It will redirect you to the directory of ChromeDriver executables files. Download as per the operating system you are currently on.



For windows, click on the "chromedriver_win32.zip" download.



The downloaded file would be in zipped format. Unpack the contents in a convenient directory.



Note: Selenium developers have defined properties for each browser that needs the location of the respective executable files to be parsed in order to invoke a browser. For example, the property defined for Chrome browser - webdriver.chrome.driver, needs the path of its executable file - D:\ChromeDriver\chromedriver.exe in order to launch chrome browser.

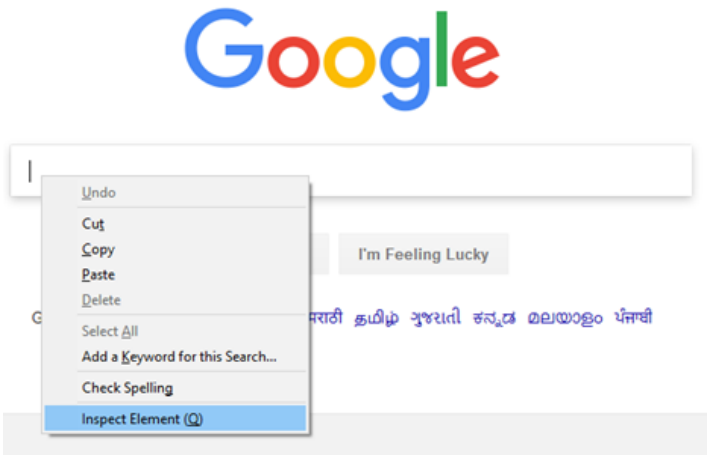
```
System.setProperty("webdriver.chrome.driver", "D:\\ChromeDriver\\chromedriver.exe");
```

Step 4:

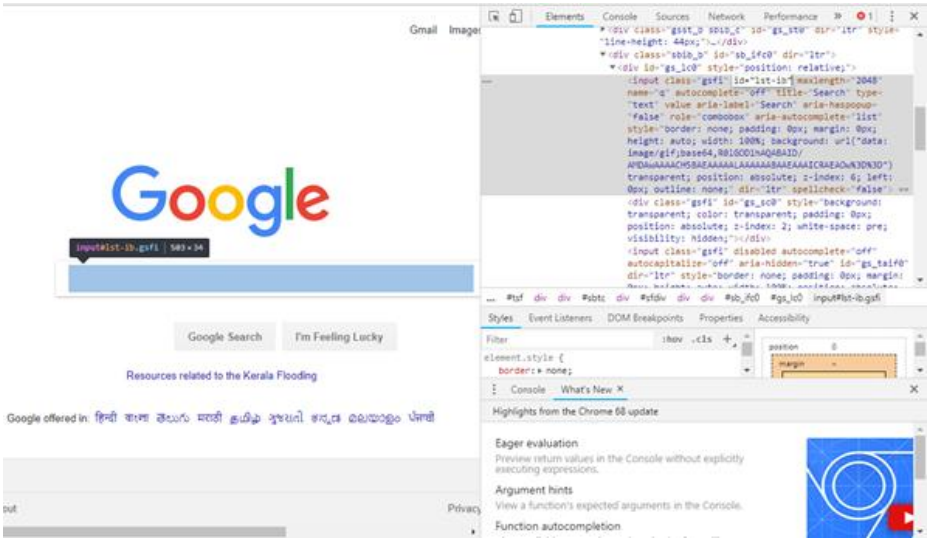
We would need a unique identification for the web elements like Google Search text box and Search button in order to automate them through our test script. These unique identifications are configured along with some Commands/Syntax to form Locators. Locators help us to locate and identify a particular web element in context of a web application.

The method for finding a unique identification element involves inspection of HTML codes.

- Open URL: https://www.google.com in your Chrome browser.
- Right click on the Google search text box and select Inspect Element.



- It will launch a window containing all the specific codes involved in the development of the test box.



- Pick the value of id element i.e. "lst-ib".

```
<input class="gsfi" id="lst-ib" maxlength="2048"
name="q" autocomplete="off" title="Search" type=
```

- Given below is the Java syntax for locating elements through "id" in Selenium WebDriver.

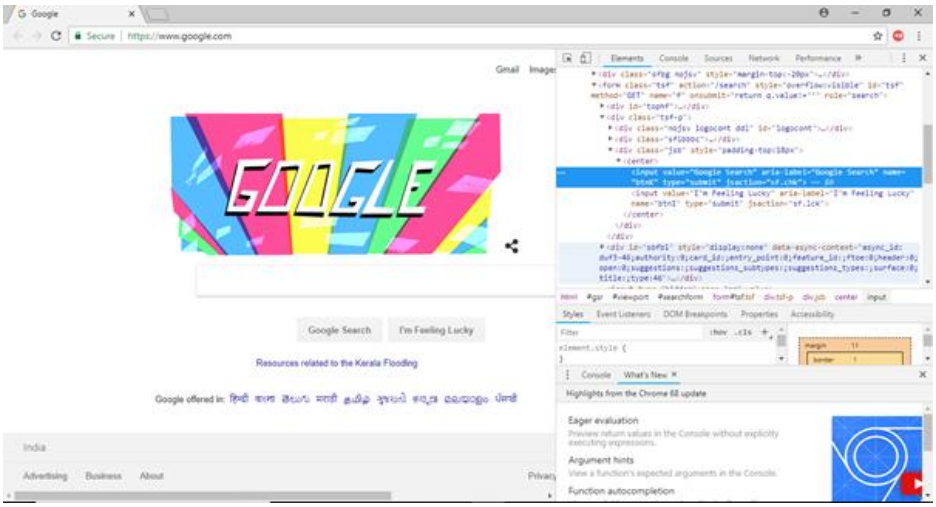
  1. driver.findElement(By.id (<element ID>))

- Here is the complete code for locating Google Search text box in our test script.

  1. driver.findElement(By.id ("lst-ib"))

- Now, right click on the Google Search button and select Inspect Element.

- It will launch a window containing all the specific codes involved in the development of the Google Search button.



- Pick the value of **name** element i.e. "btnK".

```
<input value="Google Search" aria-label="Google Search"
name="btnK" type="submit" jsaction="sf.chk"> == $0
```

- Given below is the Java syntax for locating elements through "name" in Selenium WebDriver.

  1. driver.findElement(By.name (<element name>))

- Here is the complete code for locating Google Search button in our test script.

  1. driver.findElement(By.name ("btnK"))

Step 5:

Now it is time to code. We have embedded comments for each block of code to explain the steps clearly.

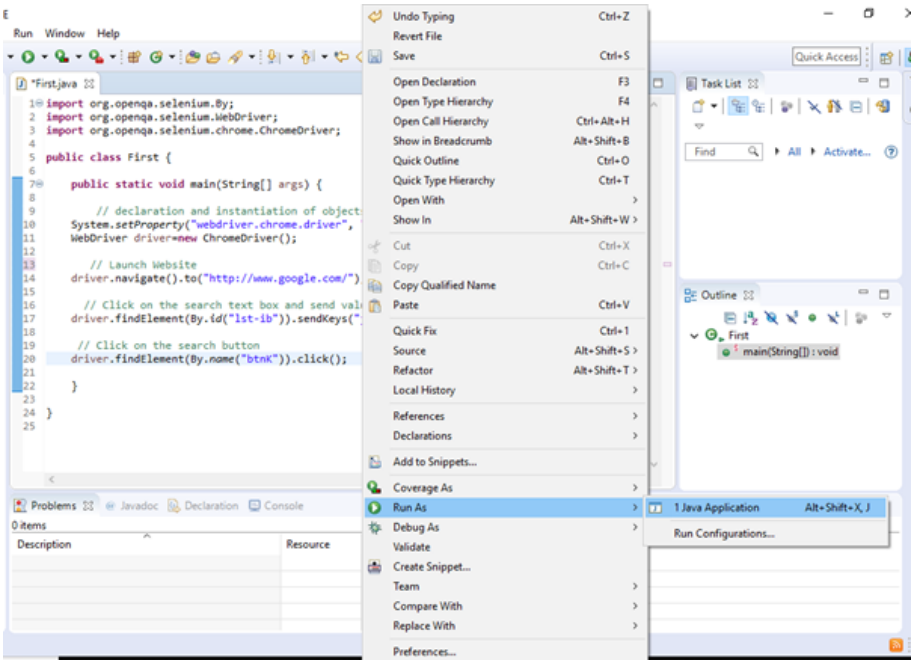
```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;


public class First {

  public static void main(String[] args) {

  // declaration and instantiation of objects/variables
```

```
    System.setProperty("webdriver.chrome.driver",
"D:\\ChromeDriver\\chromedriver.exe");
    WebDriver driver=new ChromeDriver();

// Launch website
    driver.navigate().to("http://www.google.com/");

    // Click on the search text box and send value
    driver.findElement(By.id("lst-ib")).sendKeys("javatpoint tutorials");

    // Click on the search button
    driver.findElement(By.name("btnK")).click();
    }
}
```

The Eclipse code window will look like this:

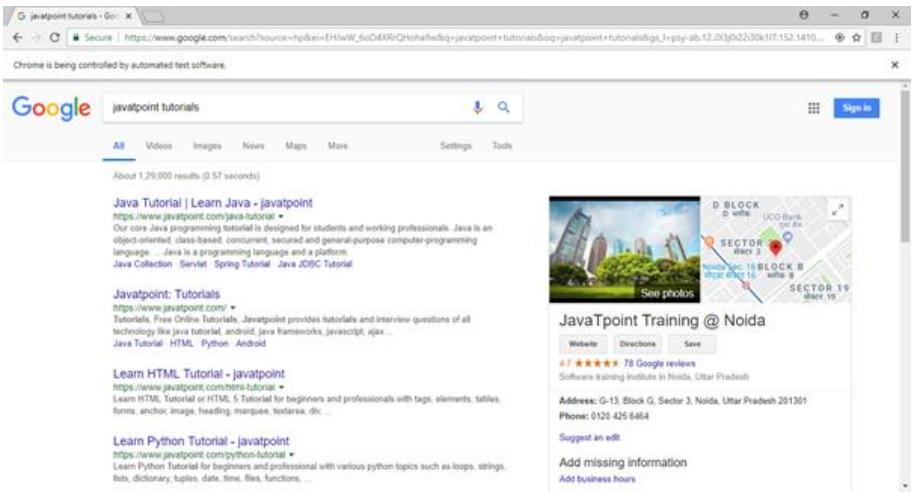
Step 6:

Right click on the Eclipse code and select **Run As > Java Application**.



Step 7:

The output of above test script would be displayed in Google Chrome browser.



- **Explanation of the Code:**

Import Packages/Statements:

In java, import statements are used to import the classes present in another packages. In simple words, import keyword is used to import built-in and user-defined packages into your java source file.

1. **org.openqa.selenium.WebDriver**: References the WebDriver interface which is required to instantiate a new web browser.

2. **org.openqa.selenium.chrome.ChromeDriver**: References the ChromeDriver class that is required to instantiate a Chrome-specific driver onto the browser instantiated by the WebDriver class.

- **Instantiating objects and variables:**

A driver object is instantiated through:

WebDriver driver=**new** ChromeDriver();

- **Launch Website:**

To launch a new website, we use navigate().to() method in WebDriver.

driver.navigate().to("http://www.google.com/");

- **Click on an element:**

In WebDriver, user interactions are performed through the use of Locators which we would discuss in later sessions of this tutorial. For now, the following instance of code is used to locate and parse values in a specific web element.

driver.findElement(By.id("lst-ib")).sendKeys("javatpoint tutorials");

4.3 UNIT END EXERCISES

Q.1) Explain selenium IDE.

Q.2) Explain steps Installation of Selenium IDE.

Q.3) Record a test case using selenium.

Q.4) Write a simple selenium WebDriver.

4.4 LIST OF REFERENCES

1) Software Testing Foundations, 4th Edition: A Study Guide for the Certified Tester Exam (Rocky Nook Computing) Fourth Edition,Andreas Spillner, Tilo Linz and Hans Schaefer.

2) Selenium WebDriver, Pearson, Rajeev Gupta, ISBN 9789332526297.

3) Selenium WebDriver Practical Guide - Automated Testing for Web

4) Applications Kindle Edition ,SatyaAvasarala ,ISBN-13: 978-1782168850

5) https://www.guru99.com/selenium-tutorial.html

6) https://www.toolsqa.com/selenium-tutorial/

7) https://www.techlistic.com/p/selenium-tutorials.html

*****

MODULE III

5

EXPERIMENT

Unit Structure

5.1 Aim

5.2 Objective

5.3 Pre-Requisite

EXPERIMENT 1

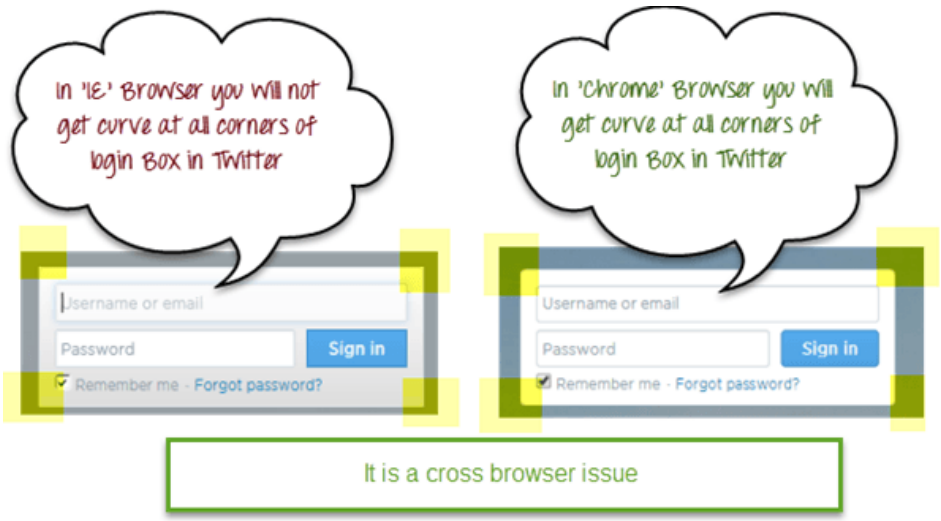
5.1 AIM

- Implementing Web Drivers on Multiple Browser i.e chrome

5.2 OBJECTIVE

Web-based applications are totally different from Windows applications. A web application can be opened in any browser by the end user. For example, some people prefer to open **https://twitter.com** in **Firefox browser,** while other's can be using **Chrome browser** or **IE**.

In the diagram below you can observe that in **IE**, the login box of Twitter is not showing curve at all corners, but we are able to see it in Chrome browser.



So we need to ensure that the web application will work as expected in all popular browsers so that more people can access it and use it.

This motive can be fulfilled with Cross Browser Testing of the product.

5.3 PRE-REQUISITE (FOR ALL EXPERIMENTS)

Selenium WebDriver installation process is completed in four basic steps:

1. Download and Install Java 8 or higher version.
2. Download and configure Eclipse or any Java IDE of your choice.
3. Download Selenium WebDriver Java Client
4. Configure Selenium WebDriver

Steps to run your Selenium Test Scripts on Chrome Browser:

Let us consider a test case in which we will try to automate the following scenarios in Google Chrome browser.

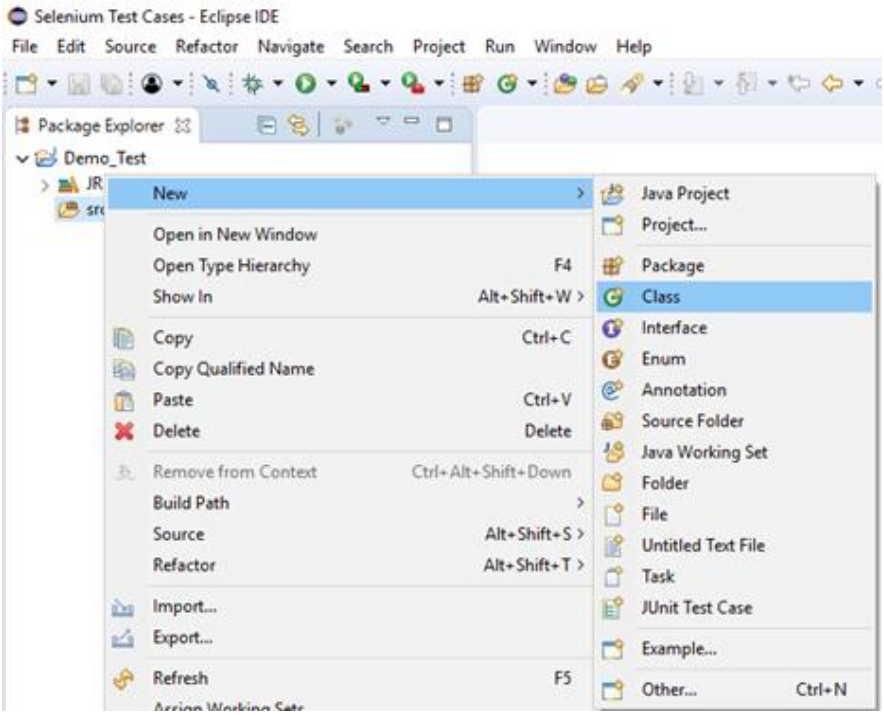
- Launch Chrome browser.
- Maximize the browser.
- Open URL: www.google.com
- Scroll down through the web page
- Click on "Core Java" link from the Java Technology section.

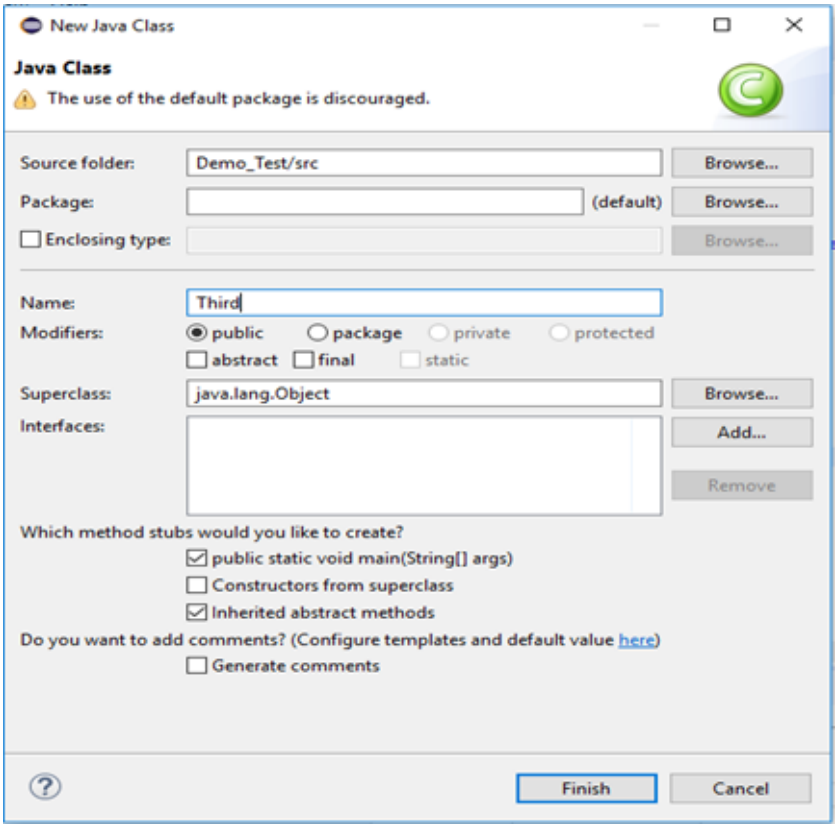
We will create our third test case in the test suite (Demo_Test).

Step 1:

Right click on the "src" folder and create a new Class File from New > Class.

Give your Class name as "Third" and click on "Finish" button.



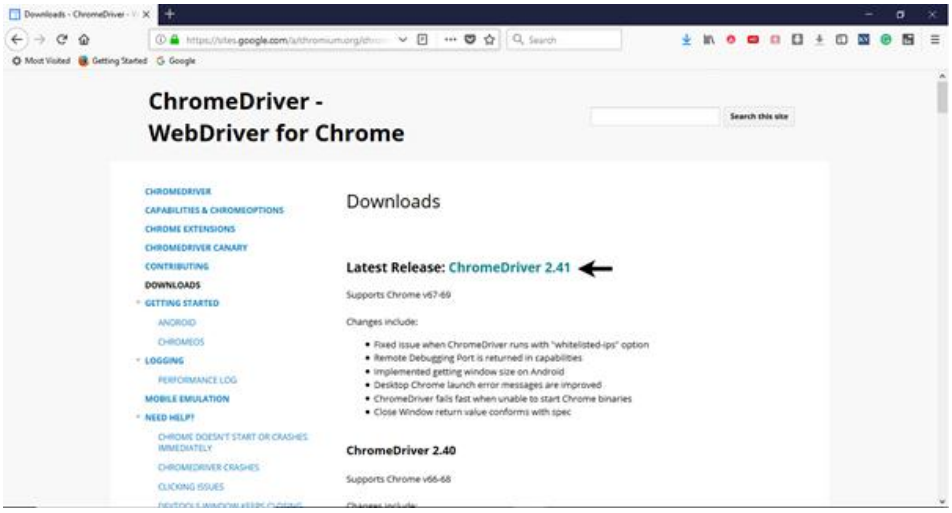


Step 2:

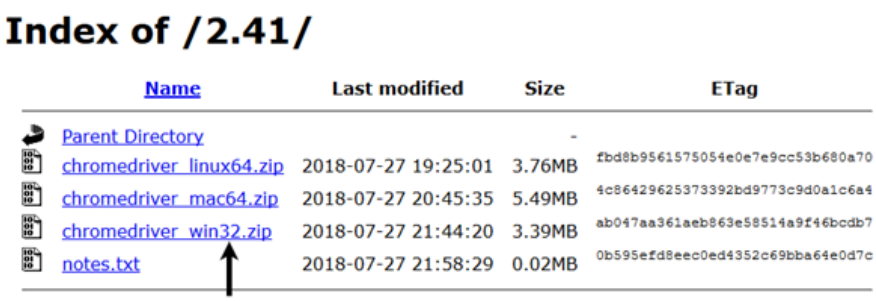
Open
URL: https://sites.google.com/a/chromium.org/chromedriver/downloads
in your browser.

Step 3:

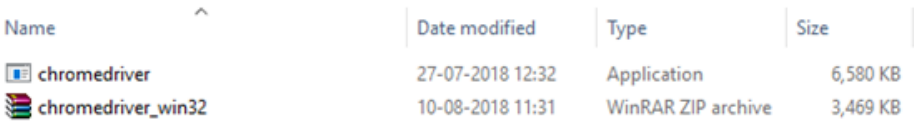
Click on the "ChromeDriver 2.41" link. It will redirect you to the directory of ChromeDriver executables files. Download as per the operating system you are working currently on.



For windows, click on the "chromedriver_win32.zip" download.



The downloaded file would be in zipped format. Unpack the contents in a convenient directory.



Step 4: Set a system property "webdriver.chrome.driver" to the path of your ChromeDriver.exe file and instantiate a ChromeDriver class.

Here is a sample code to do that.

```
// System Property for Chrome Driver

System.setProperty("webdriver.chrome.driver","D:\\ChromeDriver\\chromedriver.exe");
        // Instantiate a ChromeDriver class.
    WebDriver driver=new ChromeDriver()
```

Step5. Now it is time to code. We have embedded comments for each block of code to explain the steps clearly.(Third.java)

```
import org.openqa.selenium.By;
import org.openqa.selenium.JavascriptExecutor;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class Third {

    public static void main(String[] args) {

        // System Property for Chrome Driver

System.setProperty("webdriver.chrome.driver","D:\\ChromeDriver\\chromedriver.exe");

            // Instantiate a ChromeDriver class.
```

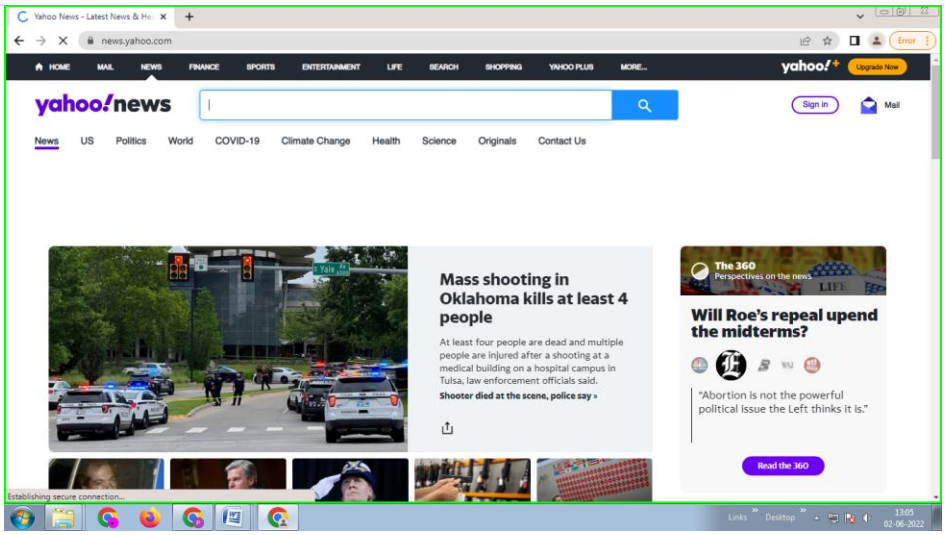
```
        WebDriver driver=new ChromeDriver();

          // Launch Website
        driver.navigate().to("http://www.news.yahoo.com/");

         //Maximize the browser
          driver.manage().window().maximize();

          //Scroll down the webpage by 5000 pixels
        JavascriptExecutor js = (JavascriptExecutor)driver;
        js.executeScript("scrollBy(0, 5000)");

         // Click on the Search button
        driver.findElement(By.linkText("Core Java")).click();

    }

}
```

Step 5:

Right click on the Eclipse code and select **Run As > Java Application**.

Step 6:

The output of above test script would be displayed in Chrome browser.



EXPERIMENT 2

AIM

- Implementing handling multiple frames

OBJECTIVE

We can handle frames in Selenium webdriver. A frame is identified with <frame> tag in the html document. A frame is used to insert an HTML document inside another HTML document.

```
<frameset frameborder="1" name="frameset-middle" cols="33%,33%,33%">
  <frame src="/frame_left" scrolling="no" name="frame-left">
    #document
      <html> ··· </html>
  </frame>
```

To work with frames, we should first understand switching between frames and identify the frame to which we want to move. There are multiple ways to switch to frames:

- **switchTo().frame(n):** The index of frame is passed as an argument to switch to. The frame index starts from 0.

Syntax:

driver.switchTo().frame(1), we shall switch to the frame with index 1.

- **switchTo().frame(name):** The frame id or name is passed as an argument to switch to.

Syntax:

driver.switchTo().frame("fname"), we shall switch to the frame with name fname.

- **switchTo.frame(webelement n):** The frame web element is passed as an argument to switch to.

Syntax:

driver.switchTo().frame(n), we shall switch to the frame with webelement n.

- **switchTo().defaultContent():** To switch back to the main page from the frame.

Syntax:

driver.switchTo().defaultContent()

Example:

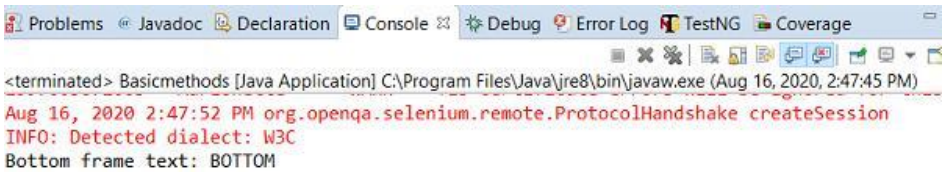
Code Implementation.

Step 1: Type following code in Eclipse:

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import java.util.concurrent.TimeUnit;
public class Framehandling{
  public static void main(String[] args) {
                                System.setProperty("webdriver.chrome.driver",
"C:\\Users\\ghs6kor\\Desktop\\Java\\chromedriver.exe");
    WebDriver driver = new ChromeDriver();
    String url = "https://the-internet.herokuapp.com/frames";
    driver.get(url);
    driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
    // identify element
    driver.findElement(By.linkText("Nested Frames")).click();
    // switch to frame with frame name and identify inside element
    driver.switchTo().frame("frame-bottom");
    WebElement l = driver.findElement(By.cssSelector("body"));
    System.out.println("Bottom frame text: " +l.getText());
    // switch to main page
    driver.switchTo().defaultContent();
    driver.quit();
  }
}
```

Step 2: run the code(Same as Experiment 1) and output will show as follows:

Output:



EXPERIMENT 3

AIM

- Implementing Selenium WebDriver - Browser Commands

OBJECTIVE

In this experiment we will learn how to implement Browser Commands using Selenium WebDriver

DESCRIPTION

Let us consider a sample test script in which will cover most of the Browser Commands provided by WebDriver.

In this sample test, we will automate the following test scenarios:

- Invoke Chrome Browser
- Open URL: https://www.google.co.in/
- Get Page Title name and Title length
- Print Page Title and Title length on the Eclipse Console
- Get page URL and verify whether it is the desired page or not
- Get page Source and Page Source length
- Print page Length on Eclipse Console.
- Close the Browser

For our test purpose, we are using the home page of "Google" search engine.

We will create our test case step by step to give you a complete understanding on how to use Browser Commands in WebDriver.

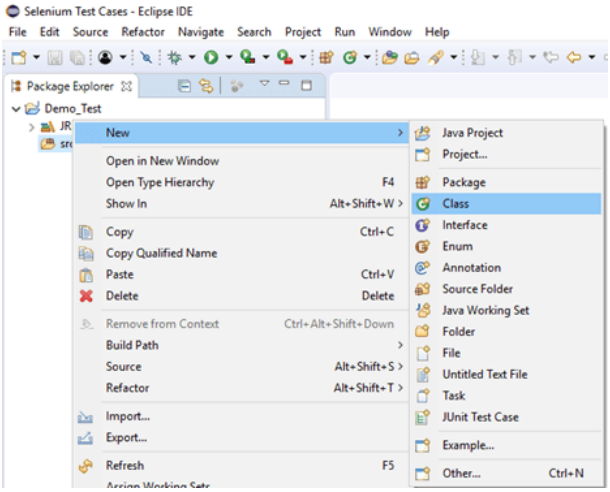
Step 1:

Launch Eclipse IDE and open the existing test suite "Demo_Test" which we have created in WebDriver Installation

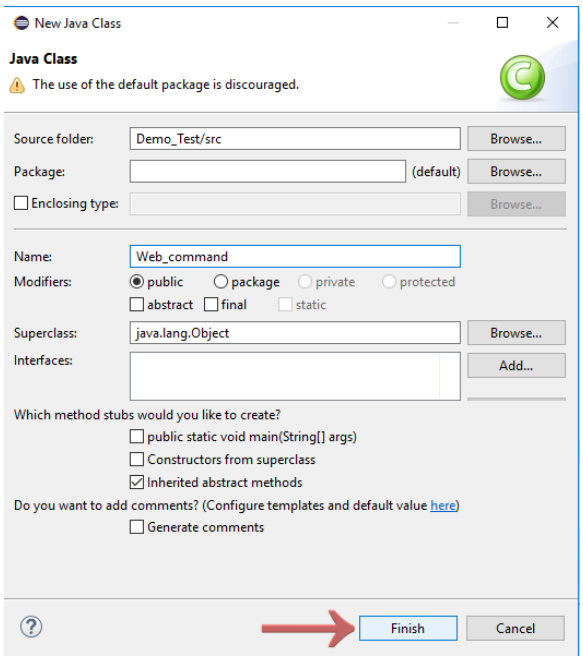
section of WebDriver tutorial.

Step 2:

Right click on the "src" folder and create a new Class File from **New > Class**.



Give your Class name as "Navigation_command" and click on "Finish" button.



Step 3:

Let's get to the coding ground.

To automate our test scenarios, first you need to know "How to invoke/launch web browsers in WebDriver?"

To invoke Google Chrome browser, we need to download the ChromeDriver.exe file and set the system property to the path of your ChromeDriver.exe file. We have already discussed this in earlier sessions of this tutorial. You can also refer to "Running test on Chrome Browser" to learn how to download and set System property for Chrome driver.

final test script will appear something like this:

(We have embedded comment in each section to explain the steps clearly)

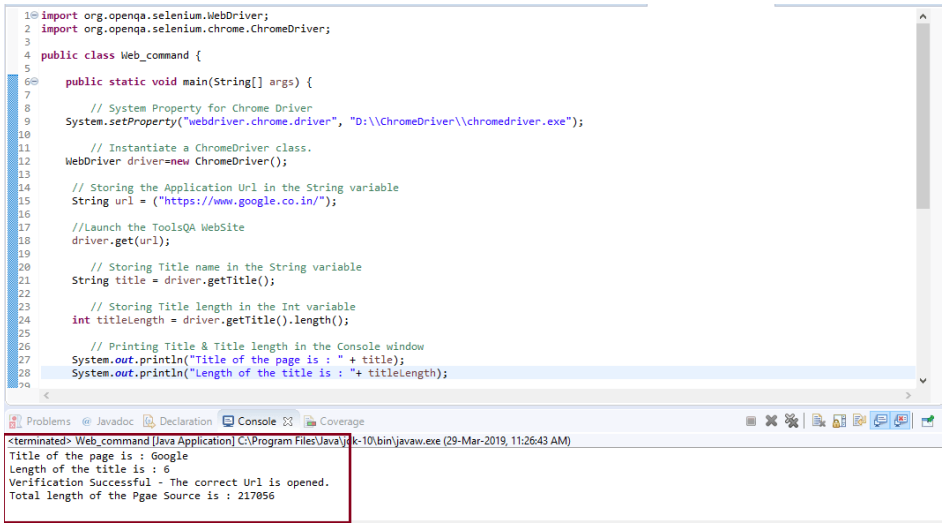
```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class Web_command {

public static void main(String[] args) {

// System Property for Chrome Driver
System.setProperty("webdriver.chrome.driver","D:\\ChromeDriver\\chrom
edriver.exe");

// Instantiate a ChromeDriver class.
WebDriver driver=new ChromeDriver();

// Storing the Application Url in the String variable
String url = ("https://www.google.co.in/");

//Launch the ToolsQA WebSite
driver.get(url);

// Storing Title name in the String variable
String title = driver.getTitle();

// Storing Title length in the Int variable
int titleLength = driver.getTitle().length();

// Printing Title & Title length in the Console window
System.out.println("Title of the page is : " + title);
System.out.println("Length of the title is : "+ titleLength);

// Storing URL in String variable
String actualUrl = driver.getCurrentUrl();

if (actualUrl.equals("https://www.google.co.in/")){
System.out.println("Verification Successful - The correct Url is opened.");
```

```
}
else{

System.out.println("Verification Failed - An incorrect Url is opened.");
        }

// Storing Page Source in String variable
String pageSource = driver.getPageSource();

// Storing Page Source length in Int variable
int pageSourceLength = pageSource.length();

// Printing length of the Page Source on console
System.out.println("Total length of the Pgae Source is : " + pageSourceLength);



//Closing browser
 driver.close();
}
}
```

To run the test script on Eclipse window, right click on the screen and click

Run as → Java application

After execution, the test script will launch the chrome browser and automate all the test scenarios. The console window will show the results for print commands.



EXPERIMENT 4

AIM

Implementing Selenium WebDriver - find element command ,Locator (id, css selector, Xpath), Input Box ,Buttons, Submit Buttons

OBJECTIVE

In this experiment we will learn how to implement - **find element command ,Locator (id, css selector, Xpath),** Input Box **,Buttons, Submit Buttons** using Selenium WebDriver

DESCRIPTION

We will see how to access these different form elements using Selenium Web Driver with Java. **Selenium encapsulates every form element as an object of WebElement.** It provides API to find the elements and take action on them like entering text into text boxes, clicking the buttons, etc. We will see the methods that are available to access each form element.

Complete Code:

Here is the complete working code

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.*;


public class Form {
    public static void main(String[] args) {
```

```
 // declaration and instantiation of objects/variables

System.setProperty("webdriver.chrome.driver","G:\\chromedriver.exe");

    WebDriver driver = new ChromeDriver();

    String baseUrl = "http://demo.guru99.com/test/login.html";

    driver.get(baseUrl);

    // Get the WebElement corresponding to the Email Address(TextField)
    WebElement email = driver.findElement(By.id("email"));


    // Get the WebElement corresponding to the Password Field

    WebElement password = driver.findElement(By.name("passwd"));


    email.sendKeys("abcd@gmail.com");
    password.sendKeys("abcdefghlkjl");
    System.out.println("Text Field Set");

    // Deleting values in the text box
    email.clear();
    password.clear();
    System.out.println("Text Field Cleared");

    // Find the submit button
    WebElement login = driver.findElement(By.id("SubmitLogin"));


    // Using click method to submit form
    email.sendKeys("abcd@gmail.com");
    password.sendKeys("abcdefghlkjl");
```

```
        login.click();
        System.out.println("Login Done with Click");


        //using submit method to submit the form. Submit used on password field
        driver.get(baseUrl);
        driver.findElement(By.id("email")).sendKeys("abcd@gmail.com");

        driver.findElement(By.name("passwd")).sendKeys("abcdefghlkjl");

        driver.findElement(By.id("SubmitLogin")).submit();

        System.out.println("Login Done with Submit");


            //driver.close();

    }
}
```

To run the above code on Eclipse window, right click on the screen and click

Run as → Java application

EXPERIMENT 5

AIM

- Demonstrate different types of alerts

OBJECTIVE

In this experiment we will learn how to implement - different types of alerts using Selenium WebDriver

DESCRIPTION

An Alert in Selenium is a small message box which appears on screen to give the user some information or notification. It notifies the user with some specific information or error, asks for permission to perform certain tasks and it also provides warning messages as well.

Alert interface provides the below few methods which are widely used in Selenium Webdriver.

1) void dismiss() // To click on the 'Cancel' button of the alert.

   driver.switchTo().alert().dismiss();

2) void accept() // To click on the 'OK' button of the alert.

   driver.switchTo().alert().accept();

3) String getText() // To capture the alert message.

   driver.switchTo().alert().getText();

4) void sendKeys(String stringToSend) // To send some data to alert box.

   driver.switchTo().alert().sendKeys("Text");

Handling Alert in Selenium Webdriver using above scenario:

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.NoAlertPresentException;
import org.openqa.selenium.Alert;


public class AlertDemo {


    public static void main(String[] args) throws NoAlertPresentException,InterruptedException  {


     System.setProperty("webdriver.chrome.driver","G:\\chromedriver.exe");
            WebDriver driver = new ChromeDriver();



    // Alert Message handling


    driver.get("http://demo.guru99.com/test/delete_customer.php");
```

```
        driver.findElement(By.name("cusid")).sendKeys("53920");

        driver.findElement(By.name("submit")).submit();

        // Switching to Alert
        Alert alert = driver.switchTo().alert();

        // Capturing alert message.
        String alertMessage= driver.switchTo().alert().getText();

        // Displaying alert message
        System.out.println(alertMessage);
        Thread.sleep(5000);

        // Accepting alert
        alert.accept();
    }

}
```

To run the above code on Eclipse window, right click on the screen and click

Run as → Java application

Note:

Use Guru99 demo site to illustrate Selenium Alert handling.

Output:

When you execute the above code, it launches the site. Try to delete Customer ID by handling confirmation alert that displays on the screen, and thereby deleting customer id from the application.

EXPERIMENT 6

AIM

- Demonstrate CheckBox and Radio Button in Selenium WebDriver

OBJECTIVE

In this experiment we will learn how to implement - CheckBox and Radio Button using Selenium WebDriver

DESCRIPTION

we will see how to identify the following form elements:

- Radio Button
- Check Box

Complete Code:

Here is the complete working code

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.*;

public class Form {
   public static void main(String[] args) {


        // declaration and instantiation of objects/variables

System.setProperty("webdriver.chrome.driver","G:\\chromedriver.exe");

      WebDriver driver = new ChromeDriver();

      driver.get("http://demo.guru99.com/test/radio.html");

      WebElement radio1 = driver.findElement(By.id("vfb-7-1"));

      WebElement radio2 = driver.findElement(By.id("vfb-7-2"));


      //Radio Button1 is selected
      radio1.click();
      System.out.println("Radio Button Option 1 Selected");


      //Radio Button1 is de-selected and Radio Button2 is selected

      radio2.click();
```

```
        System.out.println("Radio Button Option 2 Selected");


        // Selecting CheckBox
        WebElement option1 = driver.findElement(By.id("vfb-6-0"));


        // This will Toggle the Check box
        option1.click();

        // Check whether the Check box is toggled on
        if (option1.isSelected()) {
           System.out.println("Checkbox is Toggled On");


        } else {
           System.out.println("Checkbox is Toggled Off");

        }




        //Selecting Checkbox and using isSelected Method
        driver.get("http://demo.guru99.com/test/facebook.html");

        WebElement chkFBPersist =
driver.findElement(By.id("persist_box"));

        for (int i=0; i<2; i++) {

           chkFBPersist.click ();
           System.out.println("Facebook Persists Checkbox Status is -
"+chkFBPersist.isSelected());
        }
                //driver.close();


    }
}
```

To run the above code on Eclipse window, right click on the screen and click

Run as → Java application

EXPERIMENT 7

AIM

- Demonstrate synchronization in selenium(Implicit wait)

OBJECTIVE

To synchronize between script execution and application, we need to wait after performing appropriate actions in this experiment we achieve synchronization using implicit wait

DESCRIPTION

The main function of implicit Wait is to tell the web driver to wait for some time before throwing a **"No Such Element Exception".** Its default setting is knocked at zero. Once the time is set, the driver automatically will wait for the amount of time defined by you before throwing the above-given exception.

To understand how implicit wait works, let's consider an following code:

```
package JavaTpoint;
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
public class ImplicitWait{
public static void main(String[] args) throws InterruptedException
{
System.setProperty("webdriver.chrome.driver", "C:Selenium-java-
javaTpointchromedriver_win32chromedriver.exe");
WebDriver driver = new ChromeDriver();
driver.manage().window().maximize();
driver.manage().deleteAllCookies();
```

```
driver.manage().timeouts().pageLoadTimeout(40,
TimeUnit.SECONDS); // pageload timeout
driver.manage().timeouts().implicitlyWait(20, TimeUnit.SECONDS);
// Implicit Wait for 20 seconds
driver.get("https://login.google.com/");
driver.findElement(By.xpath("//input[@id='login-
username']")).sendKeys("JavaTpoint.com"); //Finding element and
sending values
Thread.sleep(1000);
driver.findElement(By.xpath("//input[@id='login-signin']")).click();
//Clicking on the next button if element is located
}
}
```

To run the above code on Eclipse window, right click on the screen and click

Run as → Java application

OUTPUT:

In the code snippet given above, the Implicit Wait is defined for only **20 seconds,** implying that the output will load or arrive within the maximum waiting time of 20 seconds for the particular element.

EXPERIMENT 8

AIM

- **Demonstrate:** Select Value from DropDown using Selenium Webdriver

OBJECTIVE

The **Select Class in Selenium** is a method used to implement the HTML SELECT tag. The html select tag provides helper methods to select and deselect the elements. The Select class is an ordinary class so New keyword is used to create its object and it specifies the web element location.

DESCRIPTION

Our use-case would follow the steps below:

- Launch the browser.
- Open "https://demoqa.com/select-menu".

- Select the Old Style Select Menu using the element id.
- Print all the options of the dropdown.
- Select 'Purple' using the index.
- After that, select 'Magenta' using visible text.
- Select an option using value.
- Close the browser

Using the methods of Select class as discussed above, the code would look like below:

```
import java.util.List;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.ui.Select;

public class DropDown {

    public static void main(String[] args) throws InterruptedException {

        //Creating instance of Chrome driver
        WebDriver driver = new ChromeDriver();

        //Step#2- Launching URL
        driver.get("https://demoqa.com/select-menu");

        //Maximizing window
        driver.manage().window().maximize();

        //Step#3- Selecting the dropdown element by locating its id
        Select select = new Select(driver.findElement(By.id("oldSelectMenu")));

        //Step#4- Printing the options of the dropdown
        //Get list of web elements
```

```
        List<WebElement> lst = select.getOptions();

        //Looping through the options and printing dropdown options
        System.out.println("The dropdown options are:");
        for(WebElement options: lst)
           System.out.println(options.getText());

        //Step#5- Selecting the option as 'Purple'-- selectByIndex
        System.out.println("Select the Option by Index 4");
        select.selectByIndex(4);
        System.out.println("Select value is: " +
select.getFirstSelectedOption().getText());

        //Step#6- Selecting the option as 'Magenta'-- selectByVisibleText
        System.out.println("Select the Option by Text Magenta");
        select.selectByVisibleText("Magenta");
        System.out.println("Select value is: " +
select.getFirstSelectedOption().getText());

        //Step#7- Selecting an option by its value
        System.out.println("Select the Option by value 6");
        select.selectByValue("6");
        System.out.println("Select value is: " +
select.getFirstSelectedOption().getText());

        driver.quit();
    }

}
```

On executing the code, you will notice that the dropdown selections are made as per the select method used, and the console window would print the options as shown below(Execution results for handling Dropdown in Selenium To run the above code on Eclipse window, right click on the screen and click

Run as → Java application)

```
The dropdown options are:
Red
Blue
Green
Yellow
Purple
Black
White
Voilet
Indigo
Magenta
Aqua
Select the Option by Index 4
Select value is: Purple
Select the Option by Text Magenta
Select value is: Magenta
Select the Option by value 6
Select value is: White
```

So this way, we can select and validate the values in a dropdown allowing single-select.

EXPERIMENT 9

AIM

- Demonstrate action classes using Selenium Webdriver(Mouse Events)

OBJECTIVE

ction Class in Selenium is a built-in feature provided by the selenium for handling keyboard and mouse events. It includes various operations such as multiple events clicking by control key, drag and drop events and many more. These operations from the action class are performed using the advanced user interaction API in Selenium Webdriver

DESCRIPTION

Below is the whole WebDriver code to check the background color of the <TR> element before and after the mouse-over.

package newproject;

import org.openqa.selenium.*;

import org.openqa.selenium.firefox.FirefoxDriver;

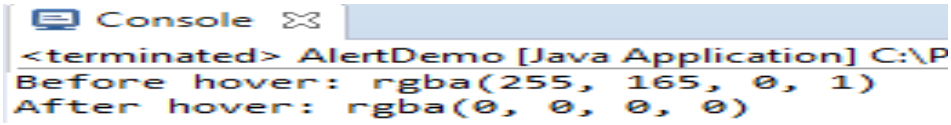
import org.openqa.selenium.interactions.Action;

import org.openqa.selenium.interactions.Actions;

```
public class PG7 {


   public static void main(String[] args) {
      String baseUrl = "http://demo.guru99.com/test/newtours/";
      System.setProperty("webdriver.gecko.driver","C:\\geckodriver.exe");
            WebDriver driver = new FirefoxDriver();


            driver.get(baseUrl);
            WebElement link_Home =
driver.findElement(By.linkText("Home"));
            WebElement td_Home = driver
                  .findElement(By
                  .xpath("//html/body/div"
                  + "/table/tbody/tr/td"
                  + "/table/tbody/tr/td"
                  + "/table/tbody/tr/td"
                  + "/table/tbody/tr"));


            Actions builder = new Actions(driver);
            Action mouseOverHome = builder
                  .moveToElement(link_Home)
                  .build();


            String bgColor = td_Home.getCssValue("background-color");
            System.out.println("Before hover: " + bgColor);
            mouseOverHome.perform();
            bgColor = td_Home.getCssValue("background-color");
            System.out.println("After hover: " + bgColor);
            driver.close();
      }
}
```

To run the above code on Eclipse window, right click on the screen and click Run as → Java application

The output below clearly states that the background color became transparent after the mouse-over.



*****

MODULE IV

6

TESTNG

Unit Structure

6.0 OBJECTIVES

After going through this unit, you will be able to:

- Understand the basic concepts in testNG
- Understand the working of testNG jar file
- Understand selenium jar Files for testNG
- How to code testcase in testNG

6.1 WHAT IS TEST NG?

TestNG is an automation testing framework in which NG stands for “Next Generation”. TestNG is inspired by JUnit which uses the annotations (@). TestNG overcomes the disadvantages of JUnit and is designed to make end-to-end testing easy.

Using TestNG, you can generate a proper report, and you can easily come to know how many test cases are passed, failed, and skipped. You can execute the failed test cases separately.

- TestNG is a very important framework when you are actually developing the framework from scratch level.

- TestNG provides you full control over the test cases and the execution of the test cases. Due to this reason, TestNG is also known as a testing framework.

- Cedric Beust is the developer of a TestNG framework.

- If you want to run a test case A before that as a pre-request you need to run multiple test cases before you begin a test case A. You can set and map with the help of TestNG so that pre-request test cases run first and then only it will trigger a test case A. In such way, you can control the test cases.

- TestNG framework came after Junit, and TestNG framework adds more powerful functionality and easier to use.

- It is an open source automated TestNG framework. In TestNG, NG stands for "Next Generation".

- TestNG framework eliminates the limitations of the older framework by providing more powerful and flexible test cases with help of easy annotations, grouping, sequencing and parametrizing.

For example:

- Suppose, you have five test cases, one method is written for each test case (Assume that the program is written using the main method without using testNG). When you run this program first, three methods are executed successfully, and the fourth method is failed. Then correct the errors present in the fourth method, now you want to run only fourth method because first three methods are anyway executed successfully. This is not possible without using TestNG.

- The TestNG in Selenium provides an option, i.e., testng-failed.xml file in test-output folder. If you want to run only failed test cases means you run this XML file. It will execute only failed test cases.

6.2 TESTNG INSTALLATION AND CONFIGURATION IN ECLIPSE

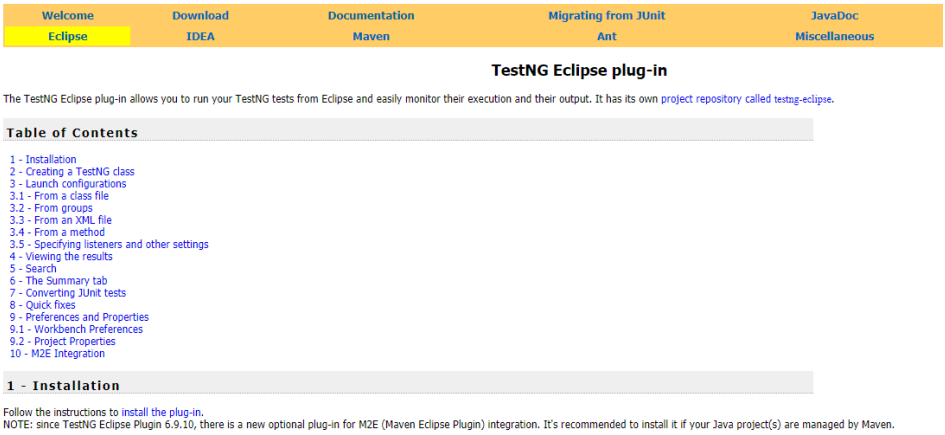
Step 1:

Go to the official website of the TestNG. Click on the link given below: https://testng.org/doc/

On clicking the above link, the screen appears as shown below:

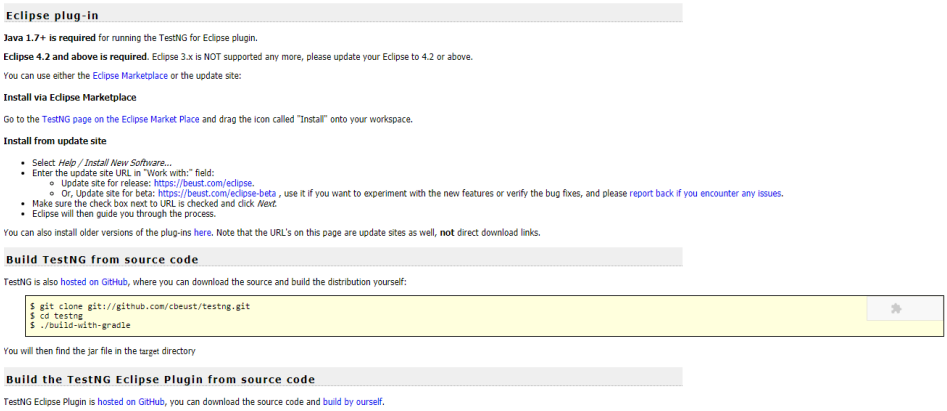
Step 2:

Click on the Eclipse appearing on the menu bar. After clicking on the Eclipse, the screen appears as shown below:



Step 3:

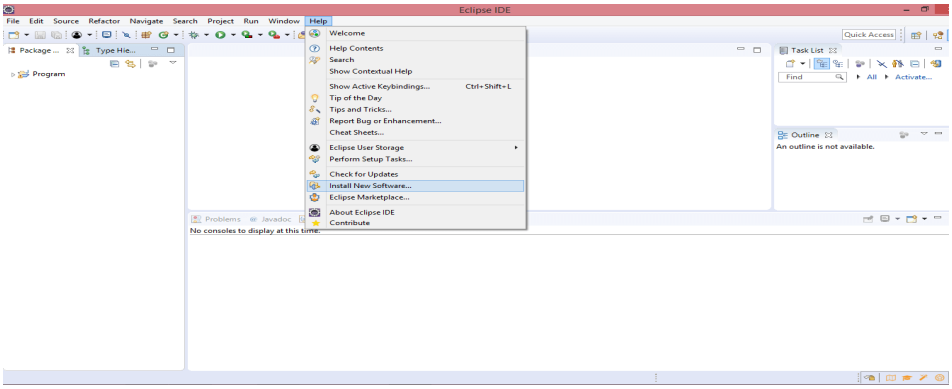
Click on the Installation appearing on the top of the Table of Contents, and then click on the "install the plug-in". On clicking on the link "install the plug in", the screen appears as shown below:



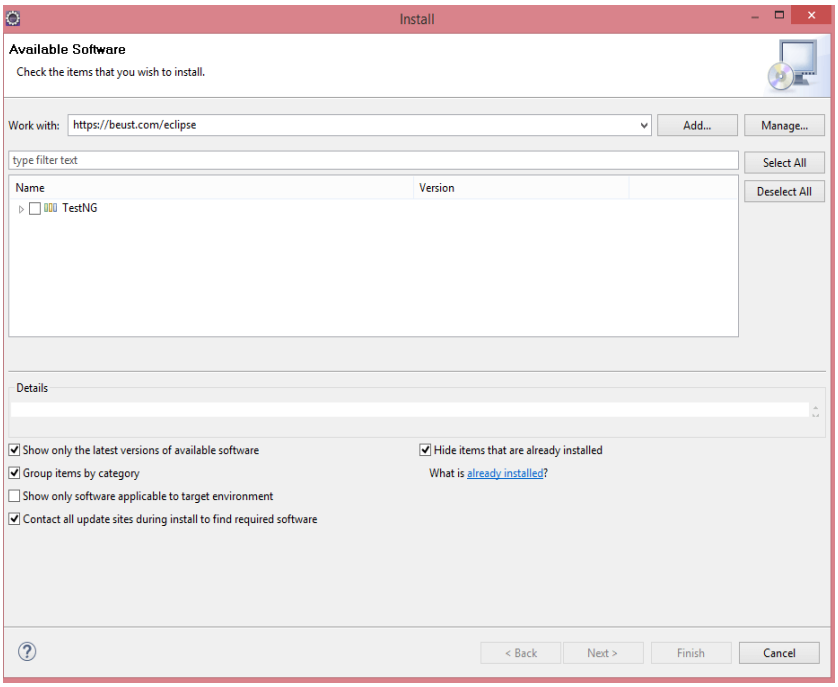
In the above screen, the URL is given https://beust.com/eclipse under the category "Install from update site". To install the TestNG plug-in in Eclipse, we need to add this given URL in Eclipse.

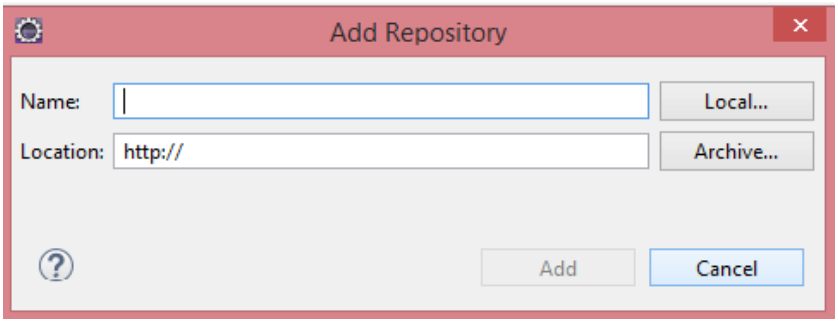
Step 4:

Open the Eclipse. Click on the Help appearing on the menu bar and then click on the Install New Software.



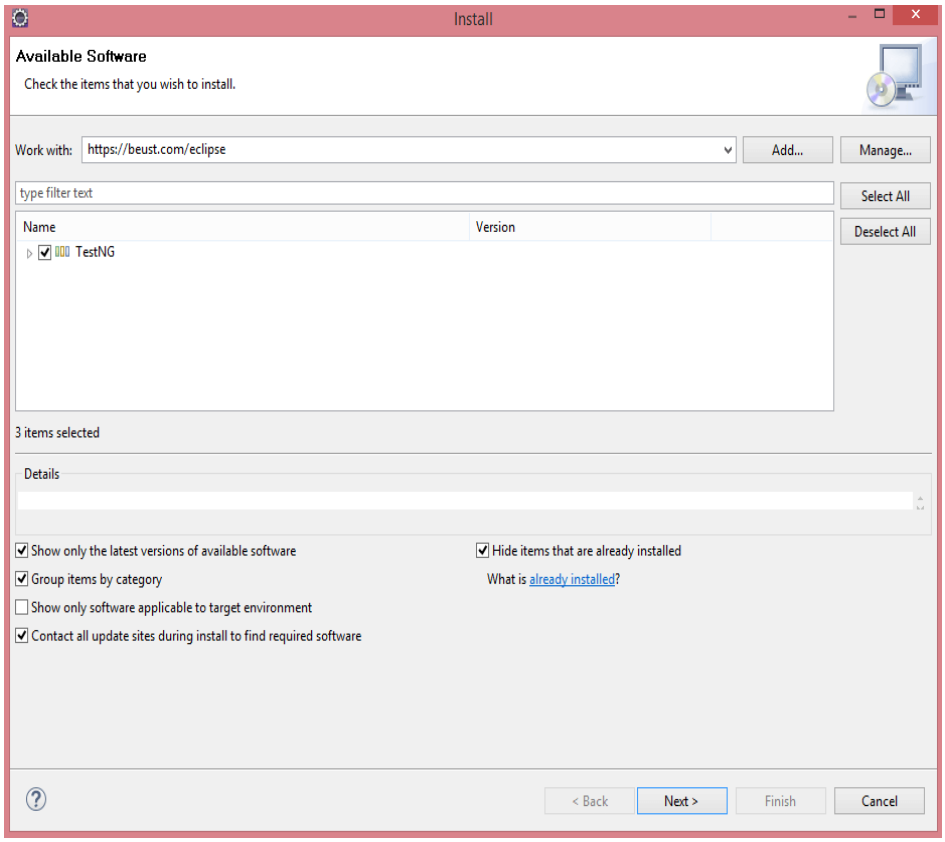
Step 5: Copy the URL https://beust.com/eclipse. Once you paste the URL, then press the Enter. However, in your case, you will see Pending for few seconds, thereafter you will see that TestNG plug-in has been loaded.





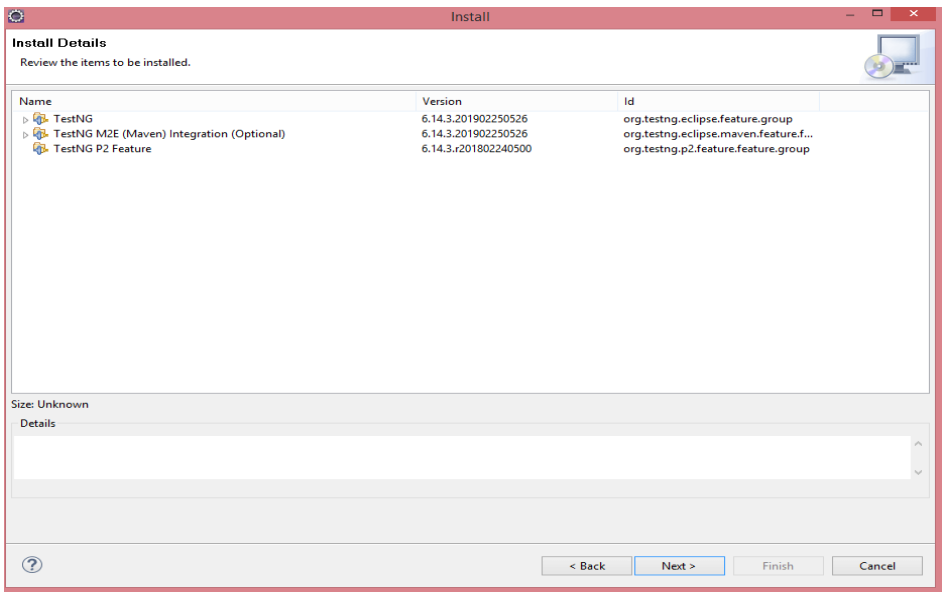
Step 6:

Click on the TestNG checkbox.



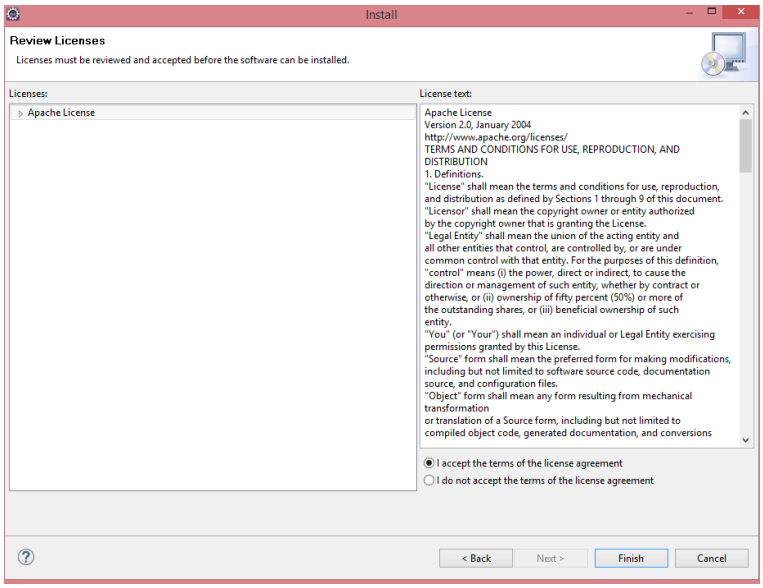
Step 7:

In the below screen, three dependencies of TestNG are shown. Now click on the Next.



Step 8:

Accept the license and then click on the Finish.



6.3 DOWNLOAD THE TESTNG JAR FILE

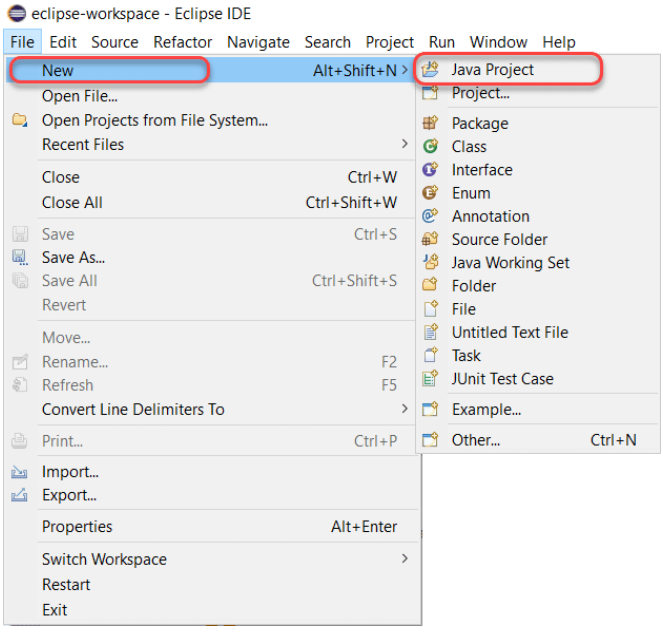
You can download the TestNG jar file from the link given below:

https://mvnrepository.com/artifact/org.testng/testng/6.7

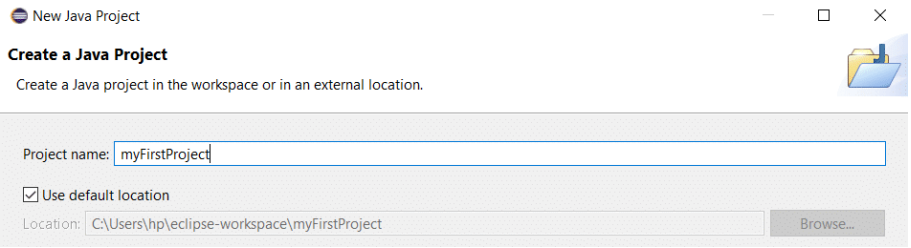
6.4 HOW TO SET UP A TESTNG TEST PROJECT IN ECLIPSE?

To set up a new TestNG project in Eclipse, open your Eclipse and follow the given steps:

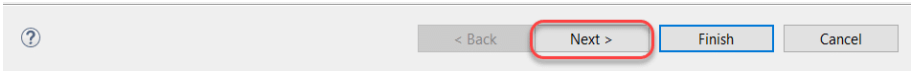
Firstly, navigate To File -> New -> Java Project.



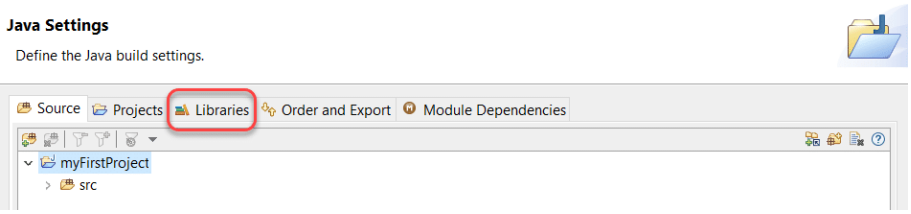
Give it a name of your choice.



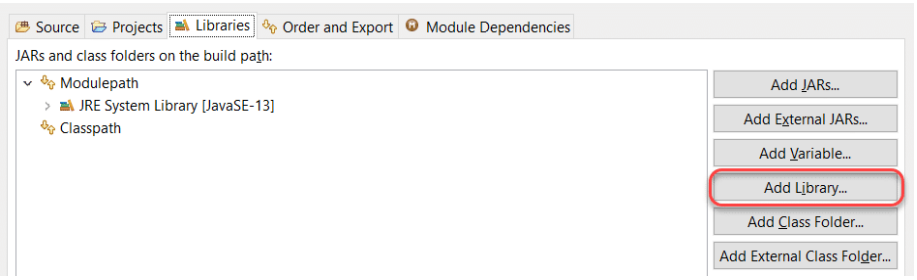
Secondly, click Next to move to the next panel.



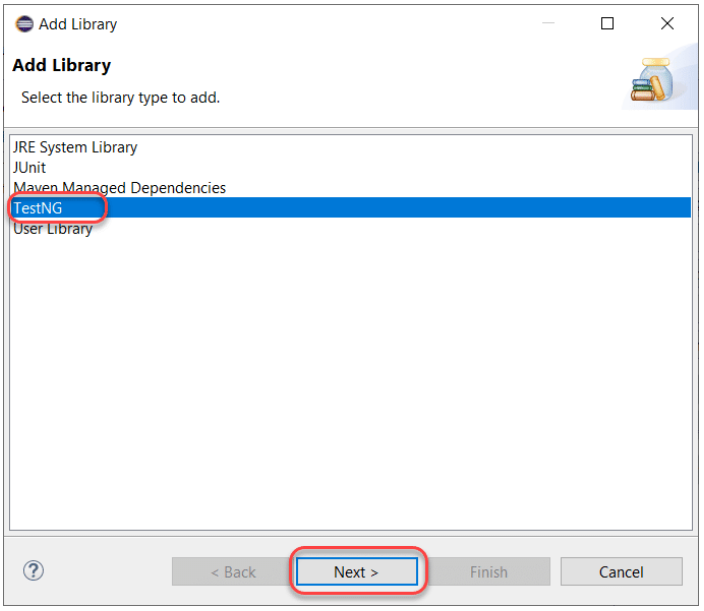
Thirdly, click on Libraries to add TestNG Libraries to your project (Only if Eclipse does not automatically add the TestNG Library).



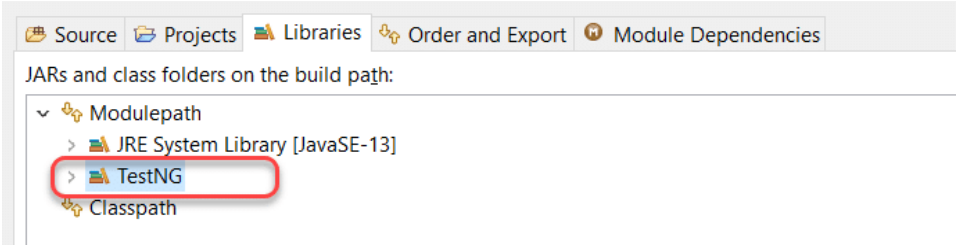
After that, select "Add Library" to add the TestNG Library.



Choose TestNG and click on Next.



Finally, click Finish to finish adding the TestNg Library in the project.



By this, we have added the TestNG Library to the project. As the next step, we need to make sure that we add the Selenium to the project before moving on to code the first test case.

6.5 HOW TO WRITE A TESTNG TEST?

Now that we are all set up with TestNG in Eclipse, we will try to write and run our first TestNG test case. But before coding our way through, we need to download Selenium Jar Files.

6.5.1 Download Selenium Jar Files for TestNG:

TestNG is majorly used with the conjunction of Selenium, so we are also going to write a TestNG test with Selenium. For that, we need to make sure that Selenium WebDriver is also set up in our system. Download the jar files from this link Download Selenium Jars.

Extract the zip file and remember the location where you extracted as we require the location in the further steps.

In the next section, we will create a TestNG class in Eclipse.

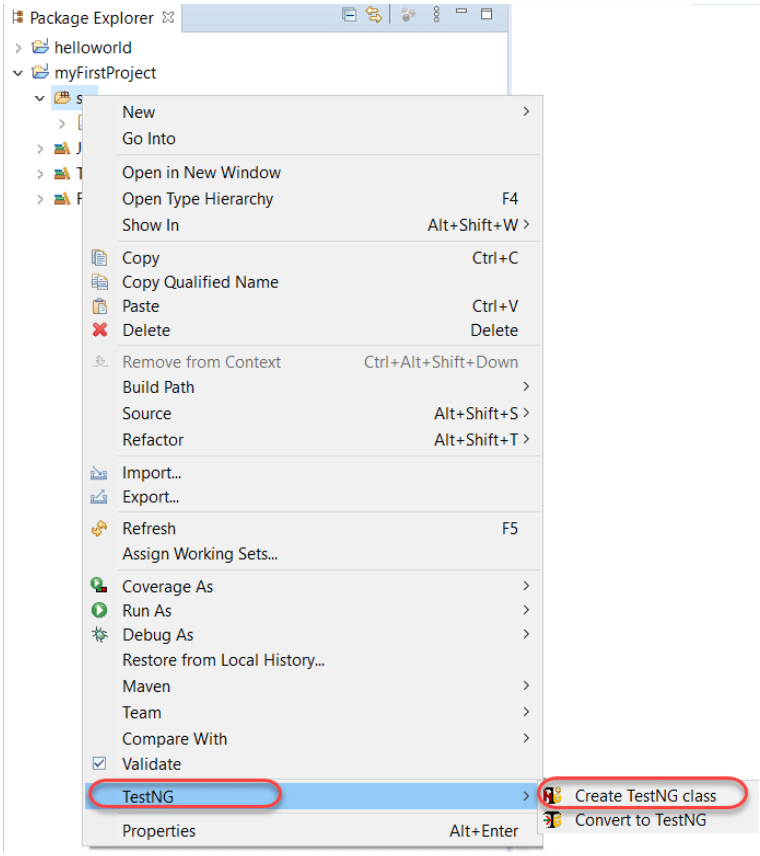
6.5.2 How to Create A TestNG Class in Eclipse:

Follow the given steps to create our first TestNG class.

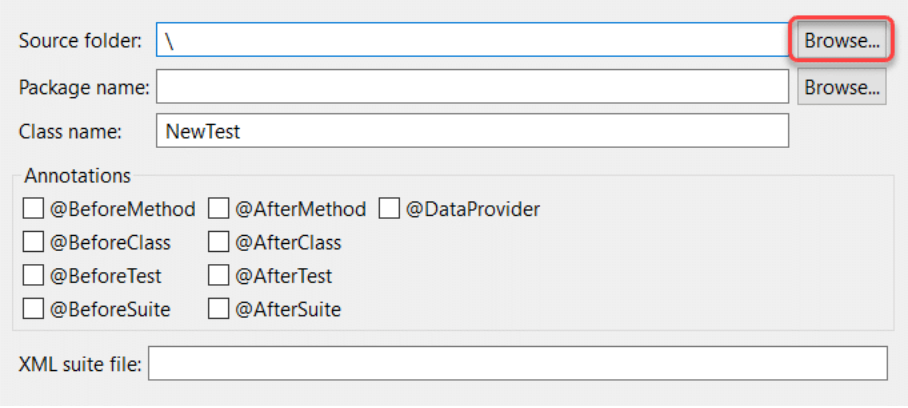
Firstly, press Ctrl+N, then select “TestNG Class” under the TestNG category and click Next.

Or

Right-click on src, go to TestNG, and select "Create TestNG Class".

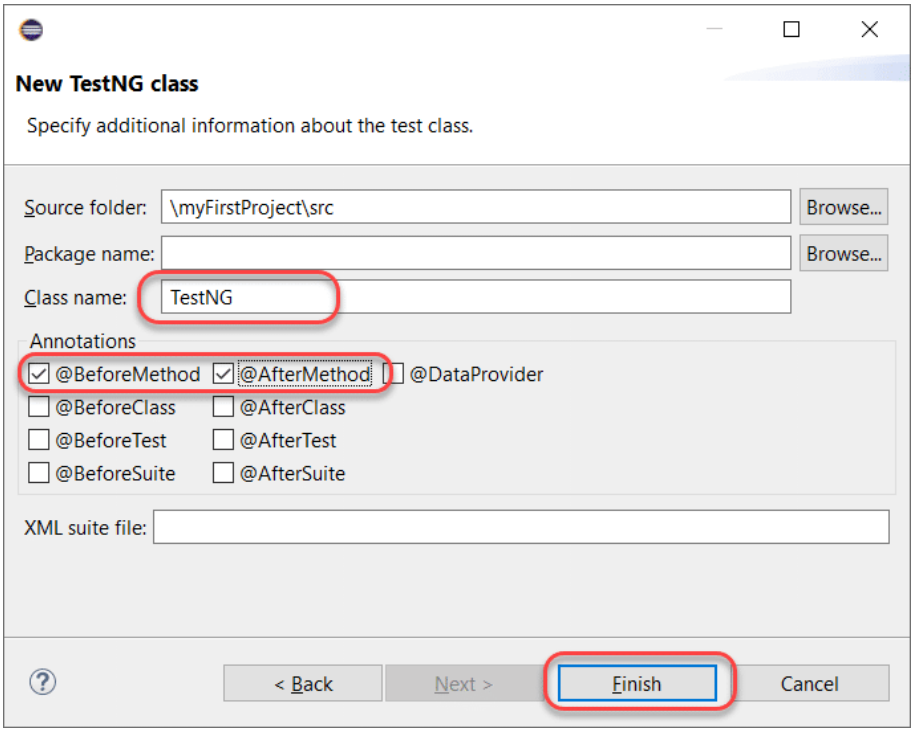


After that, the source folder name will automatically populate in the text field. But if it doesn't, like my system, you can browse your way through the src folder by clicking on the Browse button.



Thirdly, set class name as 'TestNG '

Leave the Annotations part as it is, for now, we will deal with it in the later tutorials.



Note: To know more about the TestNG Annotations, please refer to What Are TestNG Annotations?

It will display the TestNG.java test file, which is partially created for you. The test case file will contain a default method, f(), along with beforeMethod() and afterMethod() that we checked in the previous step.

```
*TestNG.java
1 import org.testng.annotations.Test;
4
5 public class TestNG {
6   @Test
7   public void f() {
8   }
9   @BeforeMethod
10  public void beforeMethod() {
11  }
12
13  @AfterMethod
14  public void afterMethod() {
15  }
16
17 }
18
19
20
```

Finally, we are all set now by creating our first test class in TestNG. We can now proceed to write the first TestNG test case.

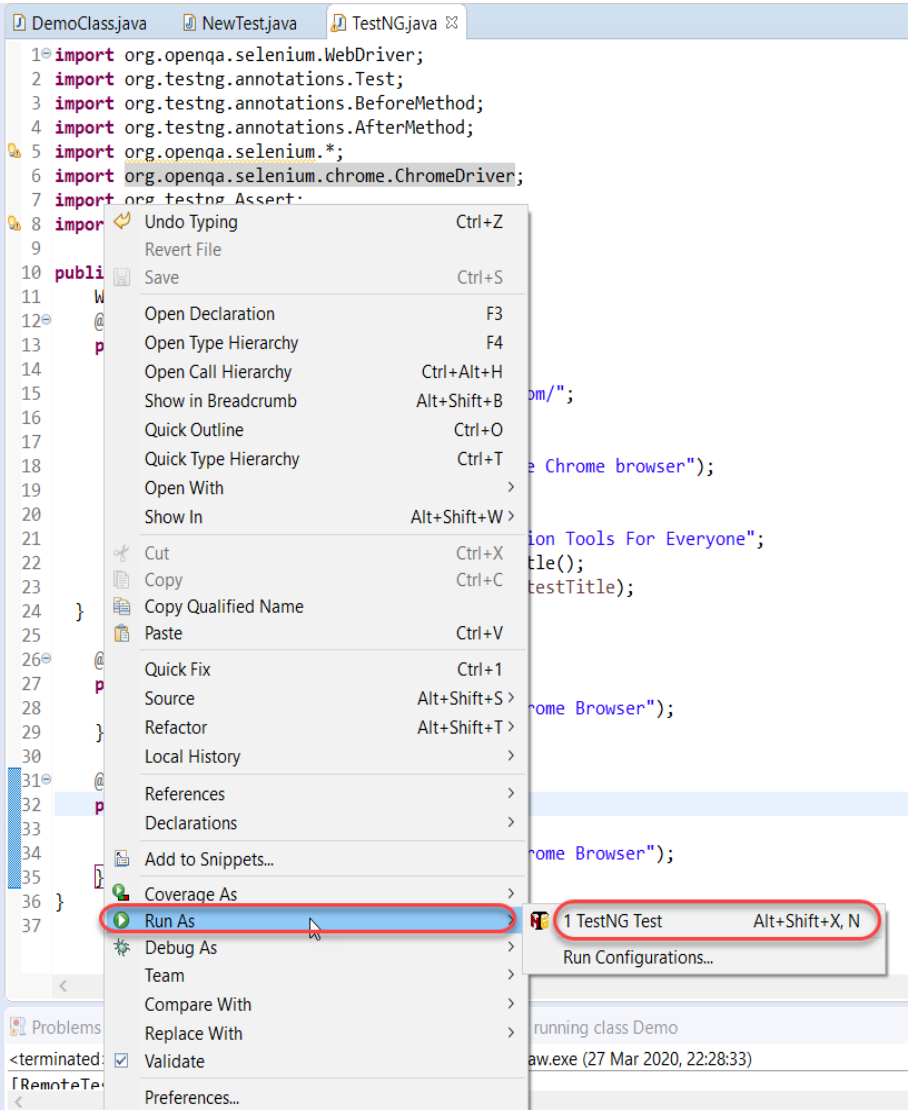
6.6 CODING OUR FIRST TEST CASE IN TESTNG

We wrote a straightforward code as a TestNG test case below for you. For understanding the Selenium part, it is recommendable to follow the Learn Selenium tutorial. Moreover, we will deal with other TestNG complexities later in the course.

You can copy and paste this code in your Eclipse.

```
import org.openqa.selenium.WebDriver;
import org.testng.annotations.Test;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.AfterMethod;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.*;

public class TestNG {
	WebDriver driver ;
	@Test
	public void f() {

	   String baseUrl = "https://www.toolsqa.com/";

		System.out.println("Launching Google Chrome browser");
		driver = new ChromeDriver();
		driver.get(baseUrl);
		String testTitle = "Free QA Automation Tools For Everyone";
		String originalTitle = driver.getTitle();
		Assert.assertEquals(originalTitle, testTitle);
 }

	@BeforeMethod
	public void beforeMethod() {
		System.out.println("Starting Test On Chrome Browser");
	}

	@AfterMethod
	public void afterMethod() {
		 driver.close();
		System.out.println("Finished Test On Chrome Browser");
	}
}
```

Right-click on the test case script and execute the test. After that, select Run As > TestNG Test.



It will run your tests successfully. We will analyze how those annotations worked in our annotations tutorial. Please note a few points concerning the above-written test case.

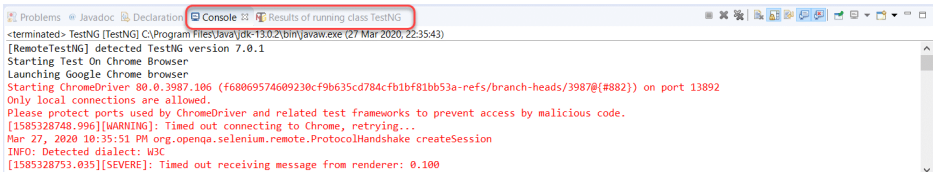
- The primary method is not necessary for a TestNG file.
- Moreover, the methods in the TestNG file need not be static in their behavior.
- In addition to the above, @Test annotations tell the underlying methods is a test method.
- Moreover, @BeforeMethod denotes that the underlying method should run before the test method.
- Similarly, @AfterMethod indicates that the underlying method should run after the test method.

6.7 HOW TO VIEW TESTNG REPORTS?

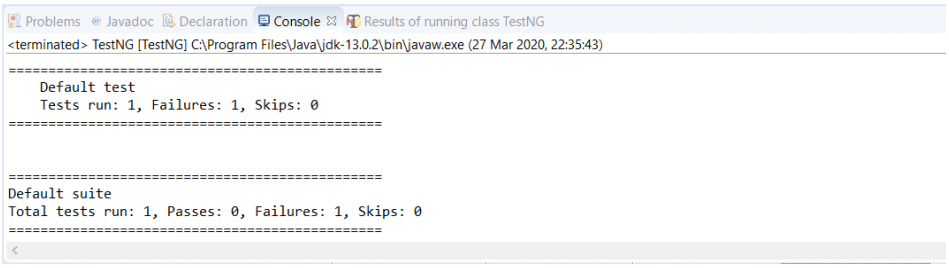
TestNG generates the reports as soon as the tests run. TestNG results are available under two sections:

- Console
- TestNG Reports

The bottom half of the screen shows both of the options.



Scrolling down in the console tab will bring the results of the tests to you.



But this is not interesting although it delivers the final aim. Additionally, for a more in-depth view of the tests, we can switch to the TestNG reports section located just beside the consol.

6.8 REFERENCES

- Software Testing Foundations, 4th Edition: A Study Guide for the Certified Tester Exam (Rocky Nook Computing) Fourth Edition,Andreas Spillner, Tilo Linz and Hans Schaefer.
- Selenium WebDriver, Pearson, Rajeev Gupta, ISBN 9789332526297.
- Selenium WebDriver Practical Guide - Automated Testing for Web Applications Kindle Edition ,SatyaAvasarala ,ISBN-13: 978-1782168850
- Testng Beginner's Guide, Packt Publishing Ltd.VarunMenon, ISBN 1782166017, 9781782166016

*****

MODULE IV

7

AUTOMATION FRAMEWORK BASICS

Unit Structure

7.0 OBJECTIVES

After going through this unit, you will be able to:

- Understand the basic concepts in Software Testing
- Understand the essential characteristics, requirements and usage of Automation Tool like Selenium Web Driver.
- Understand TestNg and Automation Framework basics
- Understand the basic concepts of quality assurance in software.

7.1 INTRODUCTION TO AUTOMATION FRAMEWORK

All software must be tested before it is made available to users. Software testing is an important part of the development cycle because it ensures that users receive high-quality deliverables. Every company focuses on testing, and the majority of them prefer automation testing to manual testing.

This is because the testing team generates various test cases and manually tests each and every feature during the manual testing process. If a defect

is discovered, it is reported to the developers so that the errors can be corrected. The process is repeated until a defect-free product is obtained.

Manual testing takes a long time. Managing the testing process is also challenging because it necessitates planning, bug tracking, and reliability analysis. Automation testing, on the other hand, uses testing tools to automate routine testing operations, making the development process faster, cheaper, and more efficient.

While automation testing is still in its early stages, businesses are looking to standardise their testing setup in order to successfully automate their legacy, desktop, web, and mobile apps with a single tool. This is where you'll find the test automation framework. It assists them in standardising all of their test automation assets, regardless of the tools they use.

7.1.1 What is the Automation Framework?

The Automation Framework is a collection of tools and processes that work together to support automated testing of any application. It combines a variety of functions, such as libraries, test data, and reusable modules.

A test automation framework is simply a set of guidelines for developing and designing test cases. It is a conceptual aspect of automated testing that assists testers in making better use of resources.

The test automation framework, rather than being an actual component of a testing software application, is a collection of concepts and tools that collaborate with items such as internal libraries and reusable code modules to provide a foundation for test automation. To make the entire process more efficient and less difficult, test automation frameworks can orient test cases by providing the test case syntax, including methodology directions, and setting up a scope for iterative testing.

There are various models for test automation frameworks, such as keyword oriented, where a table of keywords serves as the foundation for building test cases. A data-driven approach is also possible, in which the test framework provides "inputs" and then observes a series of "outputs." One way to think about it is like a graphing calculator mapping of a parabolic curve: in data-driven test cases, a variety of variables are used to examine how variable changes affect test outcomes.

7.1.2 Need for Automation Test Framework:

It is critical to use a framework for automated testing because it specifically improves the efficiency and test speed of the automation testing team. This automation testing framework also aids in improving test accuracy, as well as drastically lowering test maintenance costs and risks. Test automation has become an essential component of today's agile and DevOps practises due to the numerous benefits of automation frameworks.

In general, effective test automation supports rigorous functional tests and is more useful because it requires less design time. These automation

testing frameworks ensure greater automation consistency across the organisation and increase development activity.

The test automation framework, as previously stated, is a set of guidelines for creating and designing test cases. They are an important part of automated testing because they help testers use resources more effectively and efficiently. These automation frameworks also aid in the configuration and creation of test suites by combining various automated tests and making them executable.

Enterprises frequently face the dilemma of balancing costs and managing resources when selecting automation frameworks that can cover all of their business scenarios in order to deliver quality applications. Clearly, by selecting and implementing the best test automation framework, businesses can significantly increase the speed of test execution and the accuracy of the testing process, resulting in a higher return on investment (ROI) and higher quality products. As a result, enterprises should select effective types of automation frameworks to ensure effective application testing.

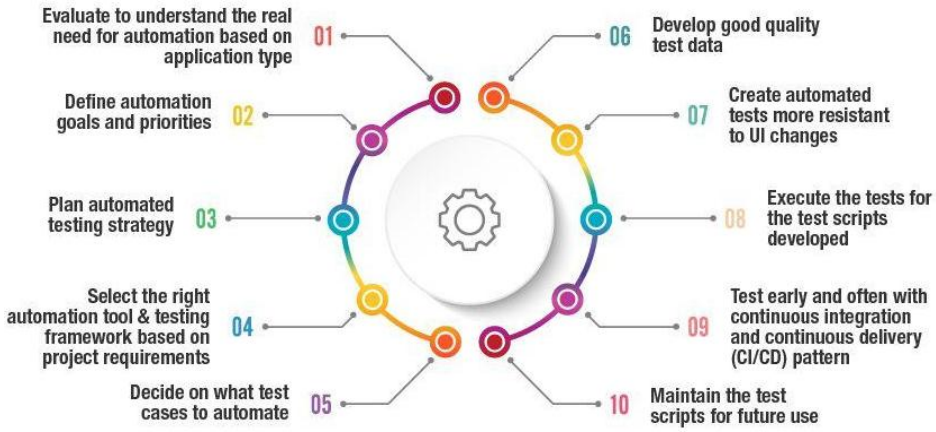


Figure: test Automation – Why, Types, Benefits approach

7.1.3 Purpose of a Test Automation Framework:

- Increases efficiency in the design and development of automated test scripts by allowing the reuse of components or code.
- Provides a structured development methodology to ensure design consistency across multiple test scripts, reducing reliance on individual test-cases.
- Enables reliable issue and bug detection and proper root-cause analysis for the system under test with minimal human intervention.
- Reduces reliance on teams by automatically selecting the test to run based on test scenarios.
- Dynamically refines test scope in response to changes in test strategy or system under test conditions.

- Increases the utilisation of various resources and allows for the greatest possible return on investment.
- Ensures an uninterrupted automated testing process with minimal human intervention.

7.2 TEST AUTOMATION FRAMEWORK TYPES

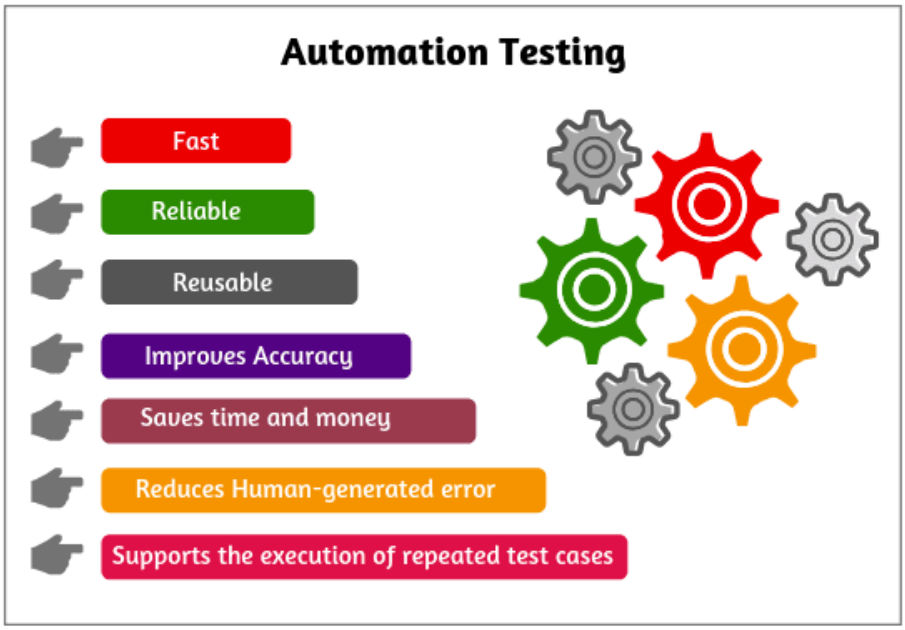


Figure: Advantages of Test Automation Framework

There are six types of test automation frameworks, each with its own architecture and set of advantages and disadvantages. When developing a test plan, it is critical to select the appropriate framework.

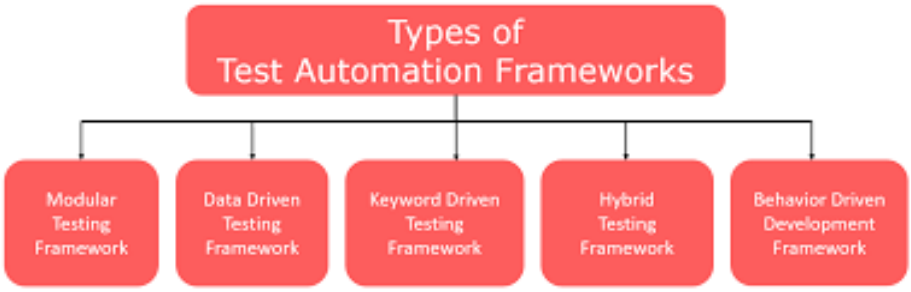


Figure: Types of Automation Framework

7.2.1 Linear Automation Framework:

A linear test automation framework, also known as a record-and-playback framework, eliminates the need for testers to write code in order to create functions, and the steps are written in a sequential order. In this process, the tester records each step, such as navigation, user input, or checkpoints, and then automatically replays the script to run the test.

Advantages of a linear framework:

- Because there is no need to write custom code, test automation expertise is not required.
- Because test scripts can be easily recorded in a short amount of time, this is one of the quickest ways to generate them.
- Because the scripts are laid out sequentially, the test workflow is easier to understand for any party involved in testing.
- This is also the quickest way to get started with automated testing, especially if you're using a new tool. Most automated testing tools today include record-and-playback capabilities, so you won't need to plan ahead of time with this framework.

Disadvantages:

- The scripts created with this framework are not reusable. Because the data is hardcoded into the test script, the test cases cannot be re-run with different sets and must be modified if the data changes.
- Maintenance is regarded as inconvenient because any changes to the application necessitate extensive rework. As the scope of testing expands, this model is not particularly scalable.

7.2.2 Modular Based Testing Framework:

To implement a modular framework, testers must divide the application under test into separate units, functions, or sections, each of which will be tested independently. After dividing the application into individual modules, a test script for each part is written and then combined to build larger tests in a hierarchical fashion. These larger groups of tests will start to represent different test cases.

Building an abstraction layer is a key strategy for using the modular framework, as it ensures that changes made in individual sections do not affect the overarching module.

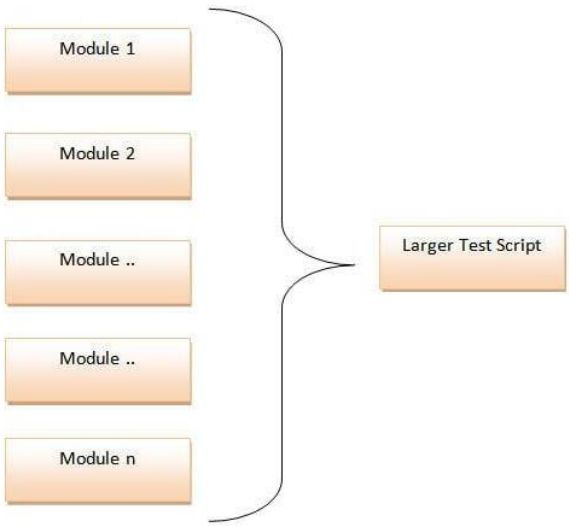


Figure: Modular Test framework

Advantages of a Modular Framework:

- If you make changes to the application, only the module and its associated individual test script will need to be fixed, which means you won't have to tinker with the rest of the application and can leave it alone.
- Because test scripts for different modules can be reused, creating test cases requires less effort.

Disadvantages of a Modular Framework:

- Because the tests are executed separately, data is still hard-coded into the test script, so multiple data sets cannot be used.
- Setting up the framework necessitates programming knowledge.

7.2.3 Library Architecture Testing Framework:

The library architecture framework for automated testing is based on the modular framework, but it has a few advantages. Rather than dividing the application under test into the various scripts that must be run, similar tasks within the scripts are identified and later grouped by function, so that the application is eventually broken down by common objectives. These functions are saved in a library and can be accessed by the test scripts whenever they are required.

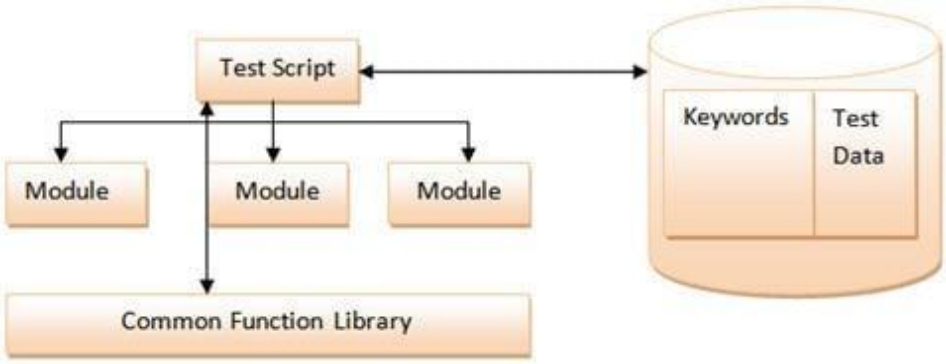


Figure: library architecture using Selenium

Advantages of a Library Architecture Testing Framework:

- Using this architecture, like the modular framework, will result in a high level of modularization, making test maintenance and scalability easier and more cost effective.
- Because there is a library of common functions that can be used by multiple test scripts, this framework has a higher degree of reusability.

Disadvantages:

- The script still has test data hard coded in it. As a result, any changes to the data will necessitate changes to the scripts.

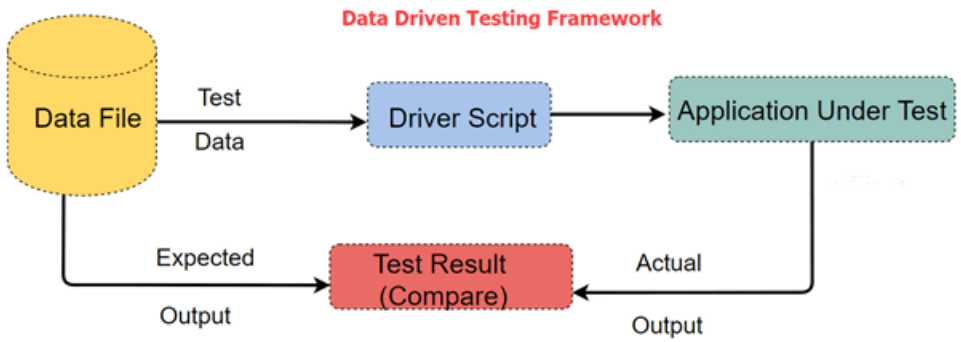
- Technical expertise is required to write and analyse the test scripts' common functions.
- Test scripts take longer to create.

7.2.4 Data-Driven Framework:

Using a data-driven framework separates test data from script logic, allowing testers to store data outside of the framework. Frequently, testers are faced with the need to test the same feature or function of an application multiple times with different sets of data. It is critical in these cases that the test data not be hard-coded in the script itself, as is the case with a Linear or Modular-based testing framework.

Setting up a data-driven test framework allows the tester to store and pass input/output parameters from an external data source, such as Excel Spreadsheets, Text Files, CSV files, SQL Tables, or ODBC repositories, to test scripts.

The test scripts are connected to the external data source and told to read and populate the necessary data when needed.



Advantages of a Data-Driven Framework:

- Multiple data sets can be used to run tests.
- By varying the data, multiple scenarios can be tested quickly, reducing the number of scripts required.
- It is possible to avoid hard-coding data so that changes to the test scripts do not affect the data being used and vice versa.
- You'll save time by running more tests more quickly.

Disadvantages:

- To properly use this framework design, you'll need a highly experienced tester who is fluent in multiple programming languages. They will have to identify and format the external data sources, as well as write code (create functions) that will seamlessly connect the tests to those external data sources.
- It takes a significant amount of time to set up a data-driven framework.

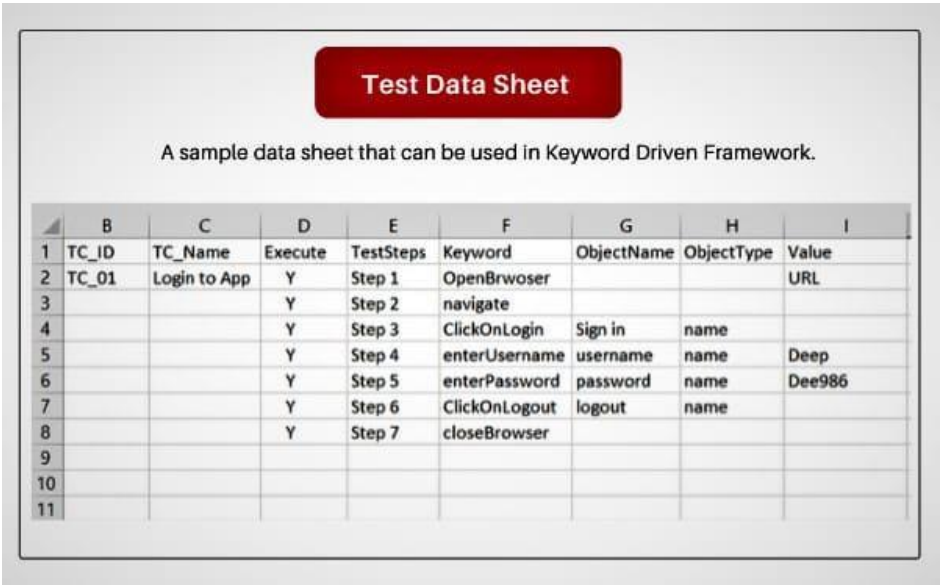
7.2.5 Keyword-Driven Framework:

Each function of the application under test is laid out in a table with a series of instructions in sequential order for each test that needs to be run in a keyword-driven framework. A keyword-driven framework separates test data and script logic in the same way that a data-driven framework does, but this approach goes a step further.

Keywords are also stored in an external data table (hence the name) in this approach, making them independent of the automated testing tool used to execute the tests. Keywords are sections of a script that represent the various actions taken to test an application's graphical user interface (GUI). These can be labelled simply as 'click' or 'login,' or more complexly as 'clicklink,' or 'verifylink.'

Keywords are stored in the table in a step-by-step fashion, each with an associated object, or the part of the UI that the action is being performed on. A shared object repository is required to map the objects to their associated actions for this approach to work properly.

Object (Application Map)	Keyword	Arguments
Textbox (Username)	Set	Test
Textbox (Password)	Set	******
Button (Login)	Click	



	B	C	D	E	F	G	H	I
1	TC_ID	TC_Name	Execute	TestSteps	Keyword	ObjectName	ObjectType	Value
2	TC_01	Login to App	Y	Step 1	OpenBrwoser			URL
3			Y	Step 2	navigate			
4			Y	Step 3	ClickOnLogin	Sign in	name	
5			Y	Step 4	enterUsername	username	name	Deep
6			Y	Step 5	enterPassword	password	name	Dee986
7			Y	Step 6	ClickOnLogout	logout	name	
8			Y	Step 7	closeBrowser			
9								
10								
11								

Figure: Keyword Driven Framework

Once the table is set up, all that remains for the testers to do is write the code that will trigger the appropriate action based on the keywords. When the test is run, the test data is read and directed to the corresponding keyword, which then executes the appropriate script.

Advantages of Keyword-Driven Frameworks:

- A basic understanding of scripting is required.
- Because a single keyword can be used in multiple test scripts, the code is reusable.
- Test scripts can be created independently of the application being tested.

Disadvantages:

- The initial cost of establishing the framework is significant. It is time-consuming and difficult. The keywords must be defined, and the object repositories / libraries must be established.
- You require an employee who is skilled in test automation.
- When scaling a test operation, keywords can be difficult to maintain. You'll need to keep working on the repositories and keyword tables.

7.2.6 Hybrid Test Automation Framework:

Automated testing frameworks, like other testing methods today, have begun to merge and overlap with one another. A hybrid framework, as the name implies, is a mix of any of the previously stated frameworks designed to capitalise on the strengths of some while mitigating the faults of others.

Every application is unique, and the techniques used to test them should be as well. As more teams adopt an agile paradigm, it is critical to have a flexible framework for automated testing. Hybrid architecture is more easily adaptable to provide the best test results.

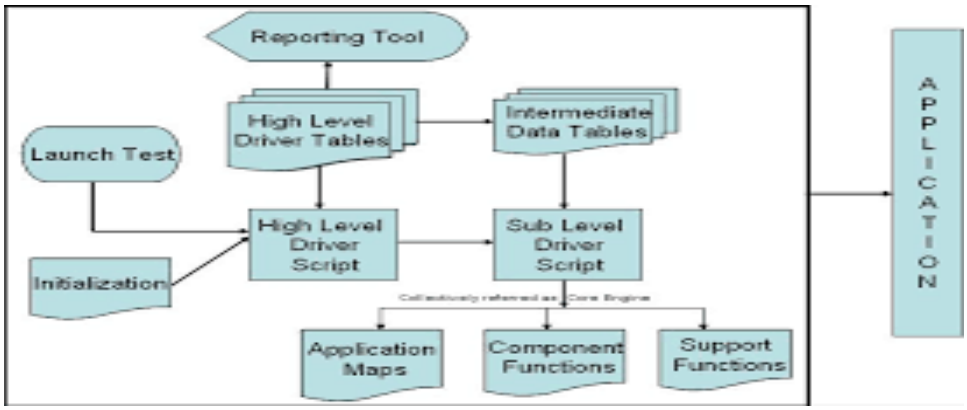
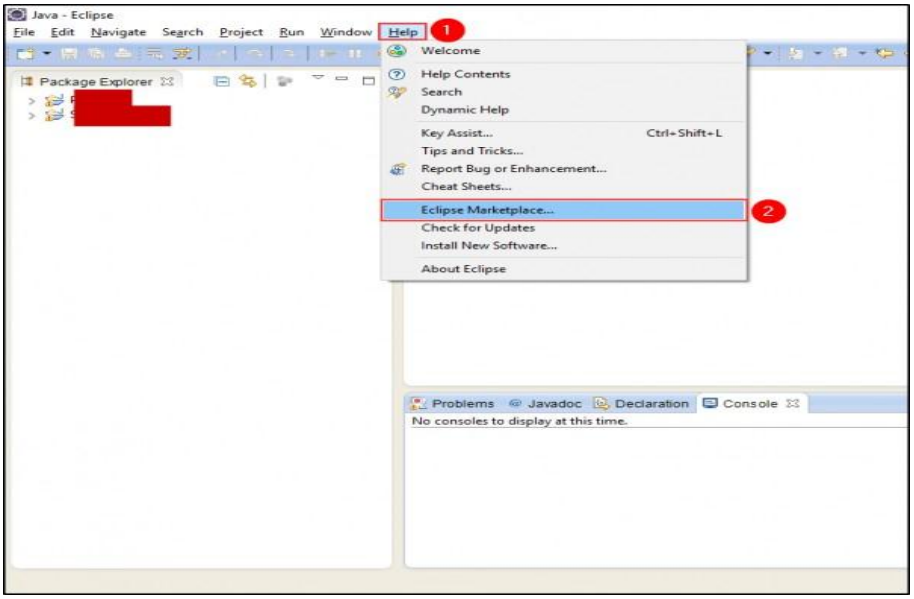


Figure: Hybrid Test Automation framework

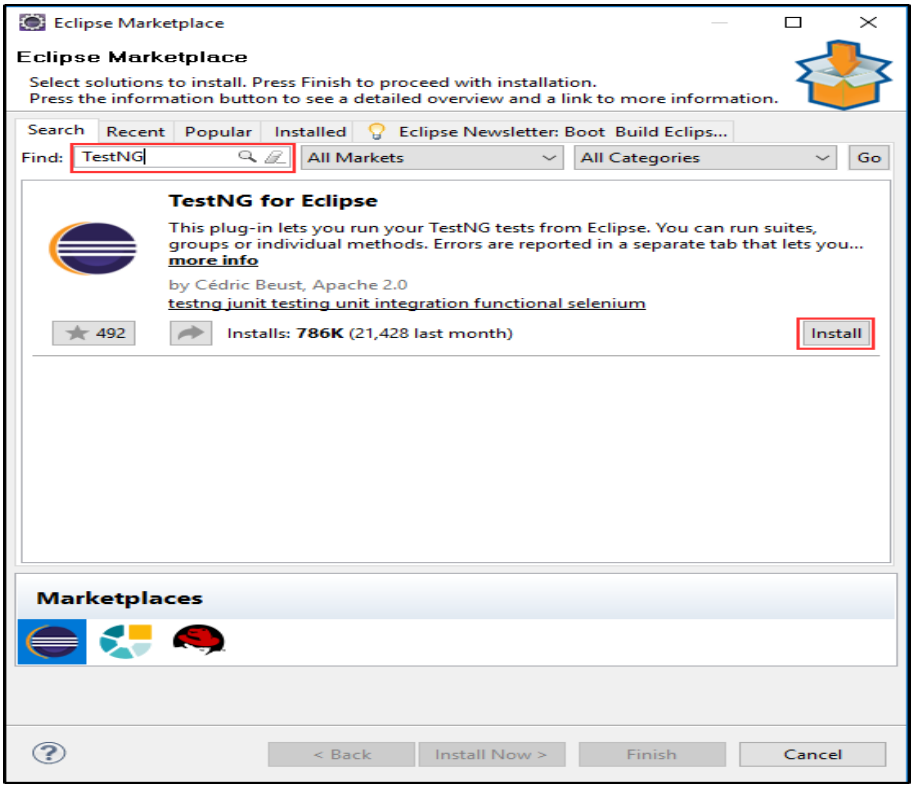
7.3 INSTALLATION OF TESTNG IN ECLIPSE

Simply open your eclipse and then follow these steps.

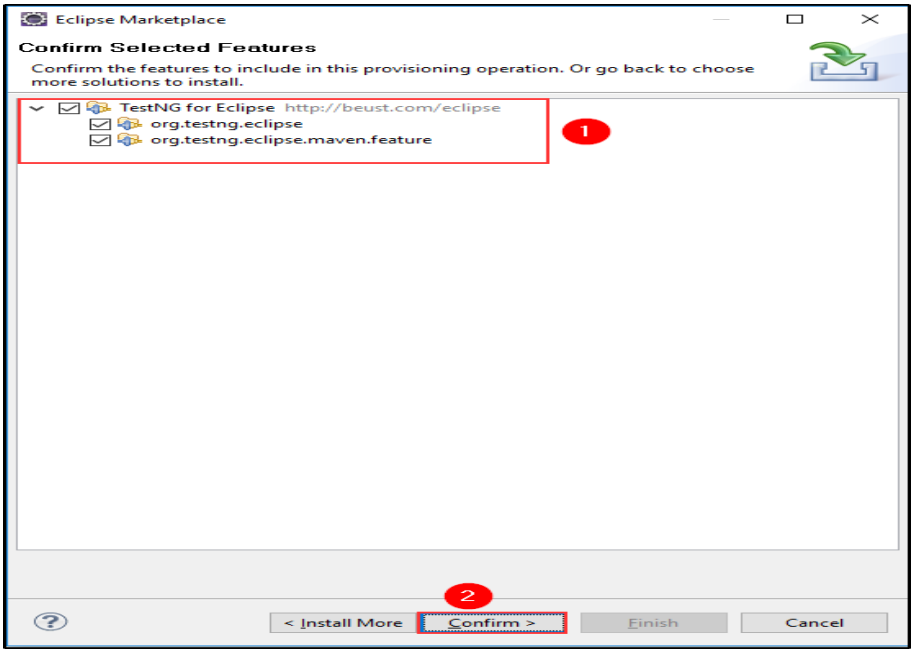
1. Click on help and then go to eclipse market place



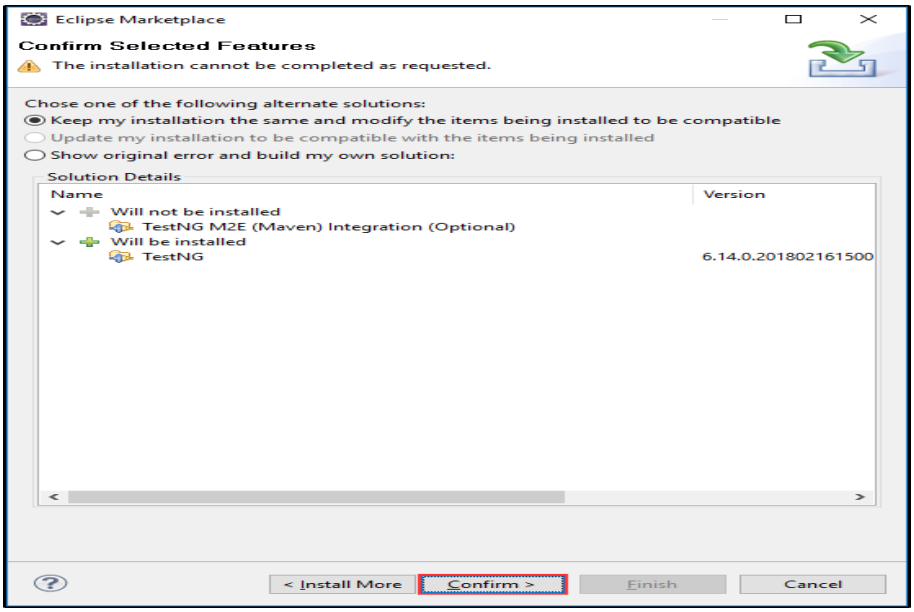
2. Search for TestNG and click on Install



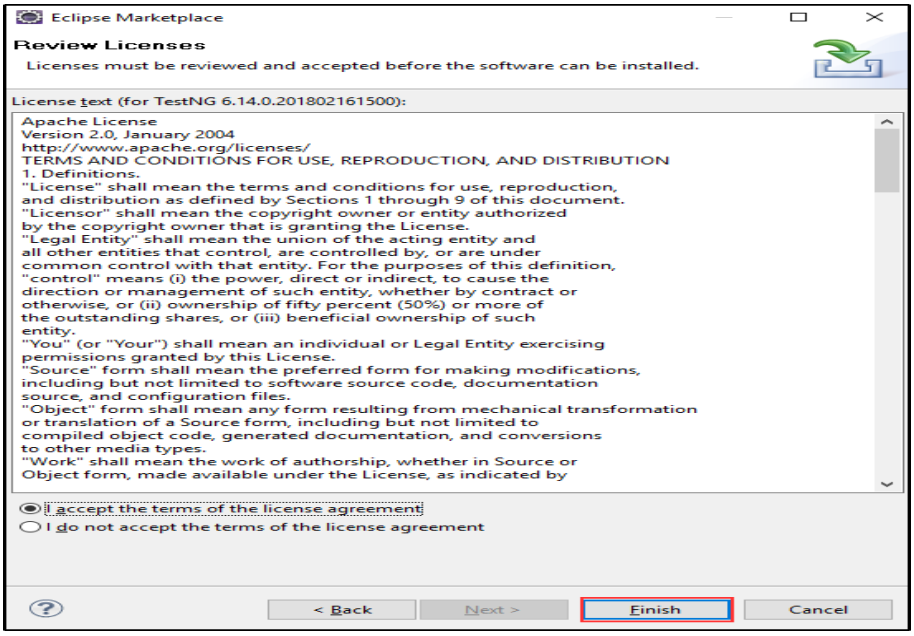
3. Select the packages that you want to Install.



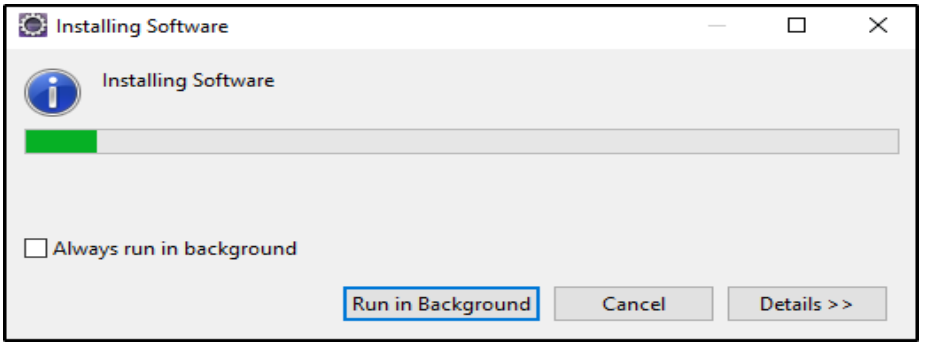
4. Confirm the installation.



5. Accept the agreement.



6. Wait for the installation to complete.



Installation successfully done!!!

7.4 REFERENCES

1. Software Testing Foundations, 4th Edition: A Study Guide for the Certified Tester Exam (Rocky Nook Computing) Fourth Edition,Andreas Spillner, Tilo Linz and Hans Schaefer.

2. Selenium WebDriver, Pearson, Rajeev Gupta, ISBN 9789332526297.

3. Selenium WebDriver Practical Guide - Automated Testing for Web Applications Kindle Edition ,SatyaAvasarala ,ISBN-13: 978-1782168850

4. Testng Beginner's Guide, Packt Publishing Ltd.VarunMenon, ISBN 1782166017, 9781782166016

7.5 WEBSITE REFERENCES

1. https://www.toolsqa.com/selenium-tutorial
2. https://www.guru99.com/selenium-tutorial.html
3. https://www.techlistic.com/p/selenium-tutorials.html

*****

MODULE V

8

QUALITY ASSURANCE

Unit Structure

8.1 PRACTICAL 1- TESTING TERMINOLOGIES AND DEFINITION

Testing:

Software testing is the act of examining the artifacts and the behavior of the software under test by validation and verification. It is a method to

check whether the actual software product matches expected requirements and to ensure that software product is defect free.

Test Cases:

Test Case is a manual approach of software testing. It is a step-by-step procedure that is used to test an application. You develop test cases to define the things that you must validate to ensure that the system is working correctly and is built with a high level of quality.

Test Suite:

A test suite is a collection of test cases that are grouped for test execution purposes.

Test Script:

Test Script is an automatic approach of software testing. Test Scripts are a line-by-line description containing the information about the system transactions that should be performed to validate the application or system under test. Test script should list out each step that should be taken with the expected results.

Test Scenarios:

A Test Scenario is a statement describing the functionality of the application to be tested. It is used for end-to-end testing of a feature and is generally derived from the use cases. It is a sequence of activities performed in a system, such as logging in, signing up a customer, ordering products, and printing an invoice. You can combine test cases to form a scenario especially at higher test levels.

Testing Strategy:

Test Strategy is a set of guidelines that explain the test design and determine how testing needs to be done.

Manual testing:

In Manual testing, testers will write test cases and test the software manually. This form of testing involves a human performing all the test steps, recording the outcome, and analyzing the results. Manual testing can be quite challenging because it often involves repeating the same set of steps many times.

Automation testing:

In Automation testing, the tester will write code and execute it to check the workflow of the application. Automated testing describes any form of testing where a computer runs the tests rather than a human. Typically, this means automated UI testing. The aim is to get the computer to replicate the test steps that a human tester would perform.

Quality:

Quality software is reasonably bug or defect free, delivered on time and within budget, meets requirements and/or expectations, and is maintainable.

Software Quality Assurance:

Software quality assurance (or SQA for short) is the ongoing process that ensures the software product meets and complies with the organization's established and standardized quality specifications.

Test Plan:

A Test Plan can be defined as a document that defines the scope, objective, and approach to test the software application. The Test Plan is a term and a deliverable. It is a document that lists all the activities in a quality assurance project, schedules them, defines the scope, roles and responsibilities, assesses risk, entry and exit criteria, determine test objective etc.

Test policy:

A document that describes how an organization runs its testing processes at a high level. It may contain a description of test levels according to the chosen life cycle model, roles and responsibilities, required/expected documents, etc.

Functional Testing:

Functional testing is a type of software testing in which the system is tested against the functional requirements and specifications. Functional testing ensures that the requirements or specifications are properly satisfied by the application. This type of testing is particularly concerned with the result of processing. It focuses on simulation of actual system usage but does not develop any system structure assumptions.

Non Functional Testing:

Non-functional testing is a type of testing to check non-functional aspects (performance, usability, reliability, etc.) of a software application. It is explicitly designed to test the readiness of a system as per non-functional parameters which are never addressed by functional testing.

A good example of non-functional test would be to check how many people can simultaneously login into a software to understand the load the system can take.

Defect:

A flaw detected in a component or system that can cause the component or system to fail to perform its required function. A defect, if encountered during execution, may cause a failure of the component or system.

Error:

A human action that produces an incorrect result.

Validation Testing:

Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment. It provides answer to the question-" Are we building the right product?"

This is carried out through activities like unit testing, integration testing, system testing and user acceptance testing.

Regression Testing:

Regression Testing is defined as a type of software testing to confirm that a recent program or code change has not adversely affected existing features. Regression Testing is nothing but a full or partial selection of already executed test cases which are re-executed to ensure existing functionalities work fine. Every time a functionality changes in a software regression testing is carried out to check if the changes made in one part of the software has not affected the other.

Ad hoc testing:

Testing carried out informally without test cases or other written test instructions. Also termed as monkey testing.

Bug:

A slang term for fault, defect, or error. Originally used to describe actual insects causing malfunctions in mechanical devices that predate computers. The International Software Testing Qualifications Board (ISTQB) glossary explains that "a human being can make an error (mistake), which produces a defect (fault, bug) in the program code, or in a document. If a defect in code is executed, the system may fail to do what it should do (or do something it shouldn't), causing a failure. Defects in software, systems or documents may result in failures, but not all defects do so." See also debugging.

8.2 PRACTICAL 2- FUNCTIONAL TESTING USING BOUNDARY VALUE ANALYSIS

Boundary-value analysis is a software testing technique in which tests are designed to include representatives of boundary values in a range. It was believed that most of the errors lie at the boundary.

In boundary value analysis, we select values on or close to boundaries and all input values may have one of the following: (i) Minimum value (ii) Just above minimum value (iii) Maximum value (iv) Just below maximum value (v) Nominal (Average) value

8.2.1 Practical 2a:

A program that accepts marks of the students between 0 and 100

There is a belief that most of the errors lie at the boundary so boundary value analysis will help us design test cases for testing the errors at the boundary.

Boundary values here are as follows:

-1, 0, 1 99 100 101

Test Inputs for test cases will be

Minimum value-1 0-1=-1

Minimum value=0

Minimum value + 1 0+1=1

Maximum value-1 100-1= 99

Maximum value 100

Maximum value +1 100+1= 101

Nominal value/Average :50

Test cases using Boundary value Analysis:

Test case id Test case input Expected output

T001	-1	Invalid Marks
T002	0	Accepted
T003	1	Accepted
T004	99	Accepted
T005	100	Accepted
T006	101	Invalid Marks
T007	50	Accepted

8.2.2 Practical 2b:

A program takes the value of age from 21 to 65. Design test cases using boundary value analysis.

BOUNDARY VALUE TEST CASE		
INVALID TEST CASE (Min Value – 1)	**VALID TEST CASES (Min, +Min, Max, -Max)**	**INVALID TEST CASE (Max Value + 1)**
20	**21, 22, 65, 64**	**66**

From the above table, we can view the following inputs that are given:

- The minimum boundary value is given as 21.
- The maximum boundary value is given as 65.
- The valid inputs for testing purposes are 21, 22, 64 and 65.
- The invalid inputs for test cases are 20 and 66.

Test Case Scenarios:

1. **Input:** Enter the value of age as 20 (21-1)

   Output: Invalid

2. **Input:** Enter the value of age as 21

   Output: Valid

3. **Input:** Enter the value of age as 22 (21+1)

   Output: Valid

4. **Input:** Enter the value of age as 65

   Output: Valid

5. **Input:** Enter the value of age as 64 (65-1)

   Output: Valid

6. **Input:** Enter the value of age as 66 (65+1)

   Output: Invalid

8.2.3 Practical 2c:

Consider a program for the determination of the largest amongst three numbers. Its input is a triple of positive integers (say x,y and z) and values are from interval [1, 300]. Design the boundary value test cases.

Solution: Invalid Value-<1, Valid value- 1-300, Invalid Value->300 or any non-numeric value.

Values to be considered for Test cases- Minimum value=1, Maximum value=300, Mininimum-1=0, Minimum+1=2, Maximum+1=301, Nominal value=150

The boundary value test cases are given below:

Test Case	x	y	z	Expected output
1.	1	150	150	150
2.	2	150	150	150
3.	150	150	150	150
4.	299	150	150	299
5.	300	150	150	300
6.	150	1	150	150
7.	150	2	150	150
8.	150	299	150	299
9.	150	300	150	300
10.	150	150	1	150
11.	150	150	2	150
12.	150	150	299	299
13.	150	150	300	300

8.2.4 Practical 2d:

Consider a program for the determination of division of a student based on the marks in three subjects. Its input is a triple of positive integers (say mark1, mark2, and mark3) and values are from interval [0, 100]. The division is calculated according to the following rules: Marks Obtained Division (Average) 75 – 100 First Division with distinction 60 – 74 First division 50 – 59 Second division 40 – 49 Third division 0 – 39 Fail Total marks obtained are the average of marks obtained in the three subjects i.e. Average = (mark1 + mark 2 + mark3) / 3 The program output may have one of the following words: [Fail, Third Division, Second Division, First Division, First Division with Distinction] Design the boundary value test cases. Solution: The boundary value test cases are given as follows :-

Solution: Invalid value-<0, Valid value-1-100, Invalid value->100 or any non-numeric value.

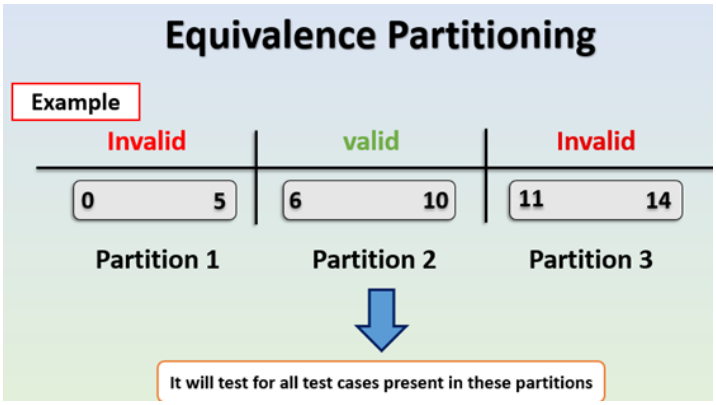
Values to be considered for Test cases- Minimum value=1, Maximum value=100, Mininimum-1=0, Minimum+1=2, Maximum+1=101, Nominal value=50. The Boundary value test cases are given below:-

Test Case	mark1	mark2	mark3	Expected Output
1.	0	50	50	Fail
2.	1	50	50	Fail
3.	50	50	50	Second Division
4.	99	50	50	First Division
5.	100	50	50	First Division
6.	50	0	50	Fail
7.	50	1	50	Fail
8.	50	99	50	First Division
9.	50	100	50	First Division
10.	50	50	0	Fail
11.	50	50	1	Fail
12.	50	50	99	First Division
13.	50	50	100	First Division

8.3 PRACTICAL 3- FUNCTIONAL TESTING USING EQUIVALENCE PARTITIONING

A large number of test cases are generated for any program. It is neither feasible nor desirable to execute all such test cases. We want to select a few test cases and still wish to achieve a reasonable level of coverage. Many test cases do not test any new thing and they just execute the same lines of source code again and again. We may divide input domain into various categories with some relationship and expect that every test case from a category exhibits the same behaviour. If categories are well selected, we may assume that if one representative test case works correctly, others may also give the same results. This assumption allows us to select exactly one test case from each category and if there are four categories, four test cases may be selected. Each category is called an equivalence class and this type of testing is known as equivalence class testing.

In case if a particular field accepts numbers only between 6 to 10 the following is the equivalence partitioning for the same:



6.3.1 Practical 3a:

3% rate of interest is given if the balance in the account is in the range of $0 to $100, 5% rate of interest is given if the balance in the account is in the range of $100 to $1000, and 7% rate of interest is given if the balance in the account is $1000 and above, we would initially identify three valid equivalence partitions and one invalid partition as shown below.

Partition 1: balance 0-100

Valid Inputs - 0-100- >=0 And <=100

Invalid Input-<0, $,#,@ A-Z

Partition 2: balance 100-1000

Valid Input - 100-1000 (>100 And <=1000)

Invalid Input- $,#,@ A-Z

Partition 3: balance >1000

Valid Input- >1000

Invalid Input- $,#,@ A-Z

Invalid	Valid partitions	Valid Partition
<0	0-100	100-1000	>1000
	3% Interest	5% Interest	7% Interest

Test Case

Test case id	**Test input** Account Balance	**Expected** Output
B001	-90	Invalid input
B002	50	3% Interest
B003	900	5% Interest
B004	2500	7% Interest
B005	A	Invalid input
B006	$	Invalid input

8.3.2 Practical 3b:

Consider the program for determination of the largest amongst three numbers. The numbers can input values between 1-100. Identify the equivalence class test cases for output and input domain.

Solution:

Input domain based equivalence classes are:

Invalid Input	Valid Input	Invalid Input
x,y,z <1	x,y,z between 1-100	x,y,z >100 or any non numeric value

Output domain equivalence classes are:

O1 = { : Largest amongst three numbers x, y, z }

O2 = { : Input values(s) is /are out of range with sides x, y, z }

The test cases are given below:

Test Case	x	y	z	Expected Output
I_1	150	40	50	150
I_2	0	50	50	Input values are out of range
I_3	50	0	50	Input values are out of range
I_4	50	50	0	Input values are out of range
I_5	101	50	50	Input values are out of range
I_6	50	101	50	Input values are out of range
I_7	50	50	101	Input values are out of range
I_8	0	0	50	Input values are out of range
I_9	50	0	0	Input values are out of range
I_{10}	0	50	0	Input values are out of range
I_{11}	301	301	50	Input values are out of range
I_{12}	50	301	301	Input values are out of range
I_{13}	301	50	301	Input values are out of range
I_{14}	0	301	50	Input values are out of range
I_{15}	301	0	50	Input values are out of range
I_{16}	50	0	301	Input values are out of range
I_{17}	50	301	0	Input values are out of range
I_{18}	0	50	301	Input values are out of range
I_{19}	301	50	0	Input values are out of range

8.4-PRACTICAL 4- FUNCTIONAL TESTING USING DECISION TABLE

Decision tables are used in many engineering disciplines to represent complex logical relationships. An output may be dependent on many input conditions and decision tables give a pictorial view of various combinations of input conditions. There are four portions of the decision table and are shown in Table 2.30. The decision table provides a set of conditions and their corresponding actions.

Table 2.30. Decision table

	Stubs	Entries
Condition	c_1 c_2 c_3	
Action	a_1 a_2 a_3 a_4	

Four Portions

1. Condition Stubs
2. Condition Entries
3. Action Stubs
4. Action Entries

Parts of the Decision Table:

The four parts of the decision table are given as:

Condition Stubs:

All the conditions are represented in this upper left section of the decision table. These conditions are used to determine a particular action or set of actions.

Action Stubs:

All possible actions are listed in this lower left portion of the decision table.

Condition Entries:

In the condition entries portion of the decision table, we have a number of columns, and each column represents a rule. Values entered in this upper right portion of the table are known as inputs. Four Portions 1. Condition Stubs 2. Condition Entries 3. Action Stubs 4.

Action Entries:

Each entry in the action entries portion has some associated action or set of actions in this lower right portion of the table. These values are known as outputs and are dependent upon the functionality of the program

8.4.1 Practical 4a:

Consider test cases based on decision table for a 'Login' Page Functionality.

Business Rules:

1. On entering correct combination of ID & Password, user should be able to login successfully.
2. User is not allowed to login when any or both of the ID & Password are incorrect /blank. In such cases, it should show 'Invalid Credentials' message.

We created the following combinations of Conditions, Actions and the respective rules in the decision table.

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8	Rule9
Conditions	TC01	TC02	TC03	TC04	TC05	TC06	TC07	TC08	TC09
UserID	Blank	Valid	Invalid	Blank	Blank	Invalid	Valid	Invalid	Valid
Password	Blank	Blank	Blank	Valid	Invalid	Valid	Invalid	Invalid	Valid
Actions									
Login Successfully									Execute
Error showing 'Invalid Credentials'	Execute	Execute	Execute	Execute	Execute	Execute	Execute	Execute	

In the above table, there are

1. conditions – UserID, Password

2. Actions – Login Successfully, Error showing 'Invalid Credentials' and

3. Options — Blank, Valid, Invalid.

So, the total number of test cases are as follows:

$$\text{Options}^{\text{conditions}} \text{ i.e } 3^2 = 9 \text{ Test cases}$$

All test cases are not valid and significant some we need to optimise the test cases

Rules 1, 2, 3, 4, and 5 cover the same action Item “Invalid Credentials” with options Blank and Invalid. Hence, we can consider any one of these test cases TC01 OR TC02 OR TC03 OR TC04 OR TC05

1. Rules 6,7, and 8 cover the same action Item “Invalid Credentials” with options Valid and Invalid. Hence, we can consider any of these test cases TC06 OR TC07 OR TC08

2. Rule 9 covers the action item “Login Successfully” with all valid options. Hence, we should consider the test case TC09.

Condensed Decision Table as shown below:

	Rule 2	Rule 8	Rule9
Conditions	TC02	TC08	TC09
UserID	Valid	Invalid	Valid
Password	Blank	Invalid	Valid
Actions			
Login Successfully			Execute
Error showing 'Invalid Credentials'	Execute	Execute	

8.4.2 Practical 4b:

Consider the problem for determining of the largest amongst three numbers where the values accepted for each is between 1 and 300. Identify the test cases using the decision table-based testing. Solution: The decision table is given as follows:

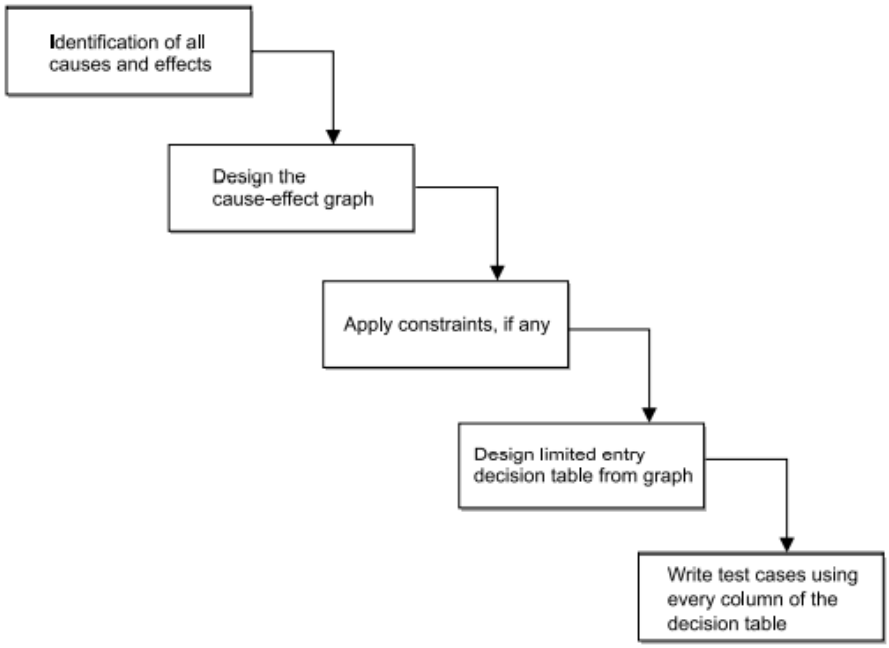
c_1: x > = 1?	F	T	T	T	T	T	T	T	T	T	T	T	T	T
c_2: x <= 300?	-	F	T	T	T	T	T	T	T	T	T	T	T	T
c_3: y > = 1?	-	-	F	T	T	T	T	T	T	T	T	T	T	T
c_4: y <= 300?	-	-	-	F	T	T	T	T	T	T	T	T	T	T
c_5: z > = 1?	-	-	-	-	F	T	T	T	T	T	T	T	T	T
c_6: z <= 300?	-	-	-	-	-	F	T	T	T	T	T	T	T	T
c_7: x>y?	-	-	-	-	-	-	T	T	T	T	F	F	F	F
c_8: y>z?	-	-	-	-	-	-	T	T	F	F	T	T	F	F
c_9: z>x?	-	-	-	-	-	-	T	F	T	F	T	F	T	F
Rule Count	256	128	64	32	16	8	1	1	1	1	1	1	1	1
a_1 : Invalid input	X	X	X	X	X	X								
a_2 : x is largest								X		X				
a_3 : y is largest											X	X		
a_4 : z is largest									X				X	
a_5 : Impossible							X							X

Test cases using Decision Table based Testing

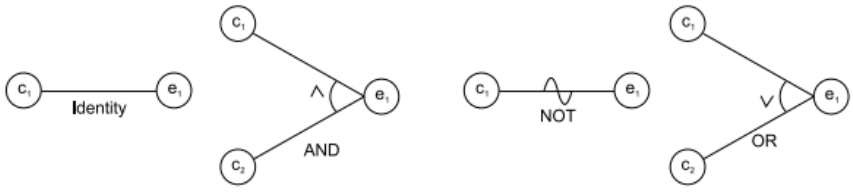
Test Case	x	y	z	Expected Output
1.	0	50	50	Invalid marks
2.	301	50	50	Invalid marks
3.	50	0	50	Invalid marks
4.	50	301	50	Invalid marks
5.	50	50	0	Invalid marks
6.	50	50	301	Invalid marks
7.	?	?	?	Impossible
8.	150	130	110	150
9.	150	130	170	170
10.	150	130	140	150
11.	110	150	140	150
12.	140	150	120	150
13.	120	140	150	150
14.	?	?	?	Impossible

8.5 PRACTICAL 5- FUNCTIONAL TESTING USING CAUSE EFFECT GRAPH

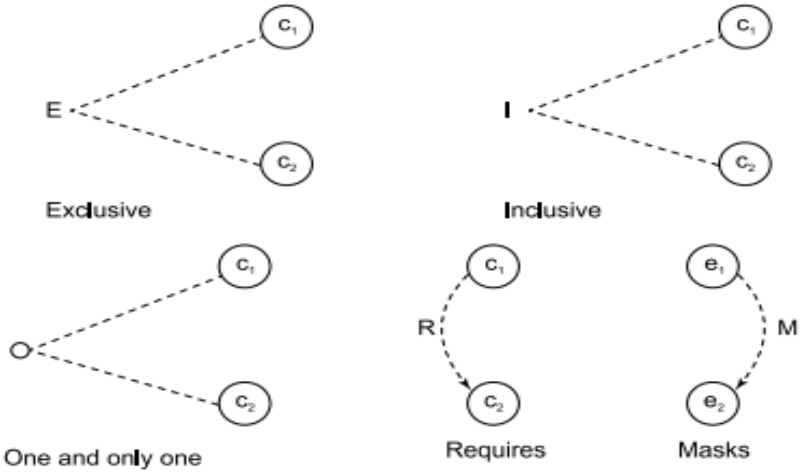
CAUSE-EFFECT GRAPHING TECHNIQUE:- This technique is a popular technique for small programs and considers the combinations of various inputs which were not available in earlier discussed techniques like boundary value analysis and equivalence class testing. Such techniques do not allow combinations of inputs and consider all inputs as independent inputs. Two new terms are used here and these are causes and effects, which are nothing but inputs and outputs respectively. The steps for the generation of test cases are given below:



Basic Notations in Cause Effect Graph:



Constraint Symbols in Cause Effect Graph:

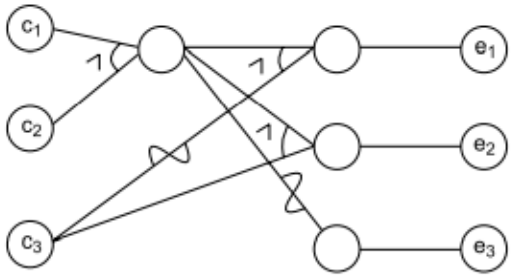


8.5.1- Practical 5a:

A tourist of age greater than 21 years and having a clean driving record is supplied a rental car. A premium amount is also charged if the tourist is on business, otherwise it is not charged. If the tourist is less than 21 year old, or does not have a clean driving record, the system will display the following message: "Car cannot be supplied" Draw the cause-effect graph and generate test cases

Solution: The causes are c1 : Age is over 21 c2 : Driving record is clean c3 : Tourist is on business and effects are e1 : Supply a rental car without premium charge. e2 : Supply a rental car with premium charge e3 : Car cannot be supplied .The Cause Effect Graph and test cases based on that are as follows :

Cause Effect Graph:



Decision Table on Rental Car Problem:

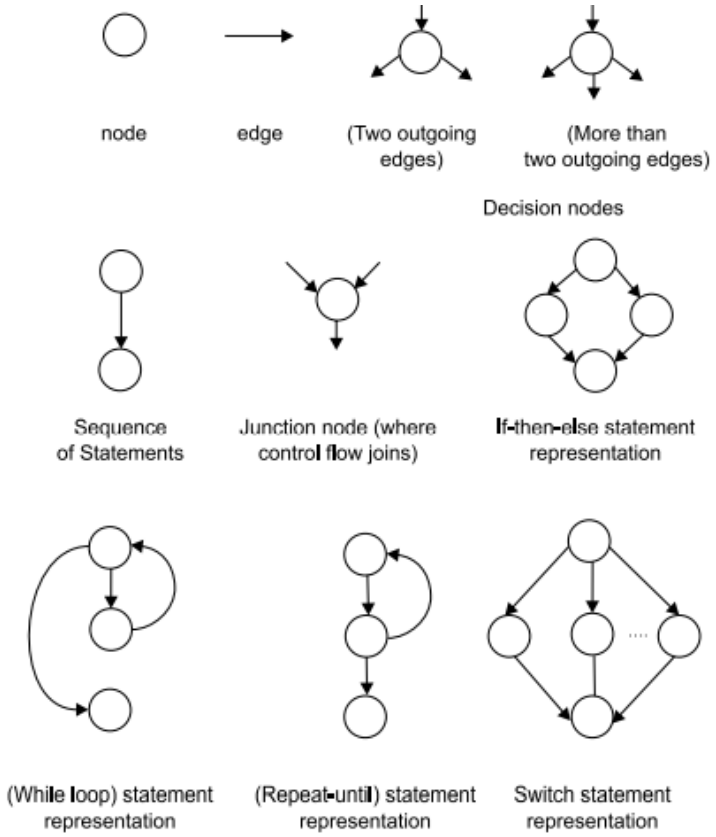
Conditions	1	2	3	4
c_1 : Over 21 ?	F	T	T	T
c_2 : Driving record clean ?	-	F	T	T
c_3 : On Business ?	-	-	F	T
e_1 : Supply a rental car without premium charge			X	
e_2 : Supply a rental car with premium charge				X
e_3 : Car cannot be supplied	X	X		

Test Cases:

Test Case	Age	Driving_record_clean	On_business	Expected Output
1.	20	Yes	Yes	Car cannot be supplied
2.	26	No	Yes	Car cannot be supplied
3.	62	Yes	No	Supply a rental car without premium charge
4.	62	Yes	Yes	Supply a rental car with premium charge.

8.6 PRACTICAL 6- PROGRAM GRAPH AND DD GRAPH TO COMPUTE CYCLOMATIC COMPLEXITY OF PROGRAM

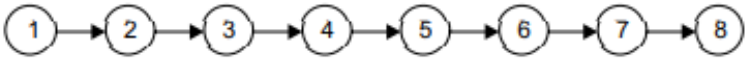
Program Graphs Program graph is a graphical representation of the source code of a program. The statements of a program are represented by nodes and flow of control by edges in the program graph. A program graph is a directed graph in which nodes are either statements or fragments of a statement and edges represent flow of control." The program graph helps us to understand the internal structure of the program which may provide the basis for designing the testing techniques. The basic constructs of the program graph are given below:



Example: Program graph to find the Square of a number

```
#include<stdio.h>
1   void main()
2   {
3   int num, result;
4   printf("Enter the number:");
5   scanf("%d", &num);
6   result=num*num;
7   printf("The result is: %d", result);
8   }
```

(a) Program to find 'square' of a number



DD Path Graphs The Decision to Decision (DD) path graph is an extension of a program graph. It is widely known as DD path graph. There may be many nodes in a program graph which are in a sequence. When we enter into the first node of the sequence, we can exit only from the last node of that sequence. In DD path graph, such nodes which are in a sequence are combined into a block and are represented by a single node. Hence, the DD path graph is a directed graph in which nodes are sequences of statements and edges are control flow amongst the nodes. All programs have an entry and an exit and the corresponding program graph has a source node and a destination node. Similarly, the DD path graph also has a source node and a destination node

Example DD Path Graph for the Program to find the Square of a number

Program graph nodes	DD path graph corresponding nodes	Remarks
1	S	Source node
2 – 7	N1	Sequential flow
8	D	Destination node

Mapping of program graph nodes and DD path graph nodes

DD Path graph

Cyclomatic Complexity This concept involves using cyclomatic number of graph theory which has been redefined as cyclomatic complexity. This is nothing but the number of independent paths through a program. McCabe [MCCA76] introduced this concept and gave three methods to calculate cyclomatic complexity.

(i) $V(G) = e - n + 2P$

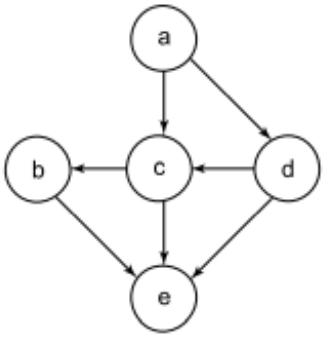
where V(G) = Cyclomatic complexity G : program graph n : number of nodes e : number of edges P : number of connected components The program graph (G) is a directed graph with single entry node and single exit node. A connected graph is a program graph where all nodes are reachable from entry node, and exit node is also reachable from all nodes. Such a program graph will have connected component (P) value equal to one. If there are parts of the program graph, the

value will be the number of parts of the program graph where one part may represent the main program and other parts may represent sub-programs.

(ii) Cyclomatic complexity is equal to the number of regions of the program graph.

(iii) Cyclomatic complexity V(G) = + 1 where is the number of predicate nodes contained in the program graph (G). The only restriction is that every predicate node should have two outgoing edges i.e. one for 'true' condition and another for 'false' condition. If there are more than two outgoing edges, the structure is required to be changed in order to have only two outgoing edges. If it is not possible, then this method (+ 1) is not applicable. Properties of cyclomatic complexity: 1. V(G) 1 2. V(G) is the maximum number of independent paths in program graph G. 3. Addition or deletion of functional statements to program graph G does not affect V(G). 4. G has only one path if V(G)=1 5. V(G) depends only on the decision structure of G.

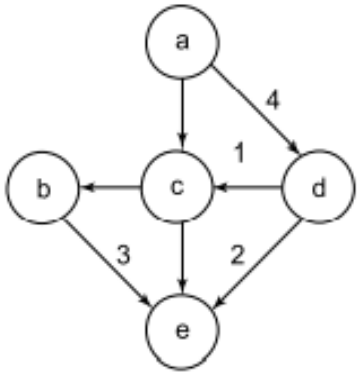
Example: Calculate the cyclomatic complexity of the following graph



The value of cyclomatic complexity can be calculated as:

(i) V(G) = e – n + 2P = 7 – 5 + 2 = 4

(ii) (ii) V(G) = No. of regions of the graph



Hence, V(G) = 4 Three regions (1, 2 and 3) are inside and 4th is the outside region of the graph

(iii) V(G) = + 1 = 3 + 1 = 4 There are three predicate nodes namely node a, node c and node d.

(iv) These four independent paths are given as:

Path 1 : ace

Path 2 : ade

Path 3 : adce

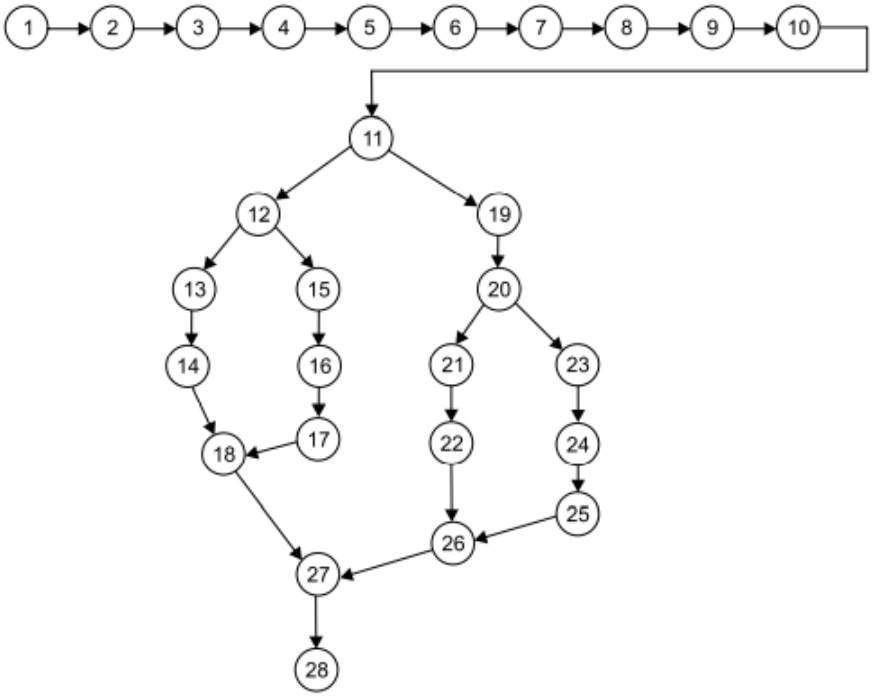
Path 4 : acbe

8.6.1 Practical 6a:

Consider the program below to find the largest among the three numbers and create the Program graph and DD graph for the same and also compute the Cyclomatic Complexity of the same

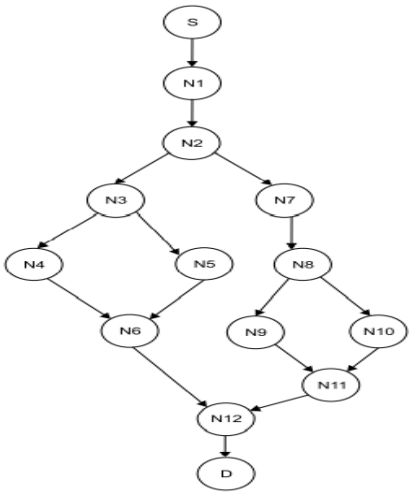
```
        #include<stdio.h>
        #include<conio.h>
1.      void main()
2.      {
3.      float A,B,C;
4.      clrscr();
5.      printf("Enter number 1:\n");
6.      scanf("%f", &A);
7.      printf("Enter number 2:\n");
8.      scanf("%f", &B);
9.      printf("Enter number 3:\n");
10.     scanf("%f", &C);
        /*Check for greatest of three numbers*/
11.     if(A>B) {
12.     if(A>C) {
13.             printf("The largest number is: %f\n",A);
14.             }
15.     else  {
16.             printf("The largest number is: %f\n",C);
17.             }
18.     }
19.     else {
20.     if(C>B) {
21.             printf("The largest number is: %f\n",C);
22.             }
```

Program Graph:



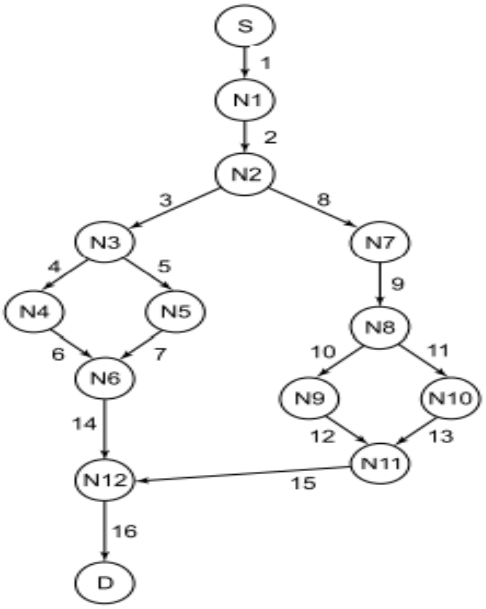
Decision to Decision Path Graph:

Program graph nodes	DD path graph corresponding node	Remarks
1	S	Source node
2 to 10	N1	Sequential nodes, there is a sequential flow from node 2 to 10
11	N2	Decision node, if true goto 12, else goto 19
12	N3	Decision node, if true goto 13 else goto 15
13, 14	N4	Sequential nodes
15, 16, 17	N5	Sequential nodes
18	N6	Junction node, two edges 14 and 17 are terminated here
19	N7	Intermediate node terminated at node 20
20	N8	Decision node, if true goto 21 else goto 23
21, 22	N9	Sequential nodes
23, 24, 25	N10	Sequential nodes
26	N11	Junction node, two edges 22 and 25 are terminated here
27	N12	Junction node, two edges 18 and 26 are terminated here
28	D	Destination node



Cyclomatic Complexity of the same graph:

Find the independent path



Solution:

(i) V(G) = e – n + 2P = 16 – 14 + 2 = 4

(ii) V (G) = Number of Regions = 4

(iii) V (G) = + 1 = 3 (N2, N3, N8) + 1 = 4

There are 4 independent paths as given below:

(a) S, N1, N2, N3, N4, N6, N12, D

(b) S, N1, N2, N3, N5, N6, N12, D

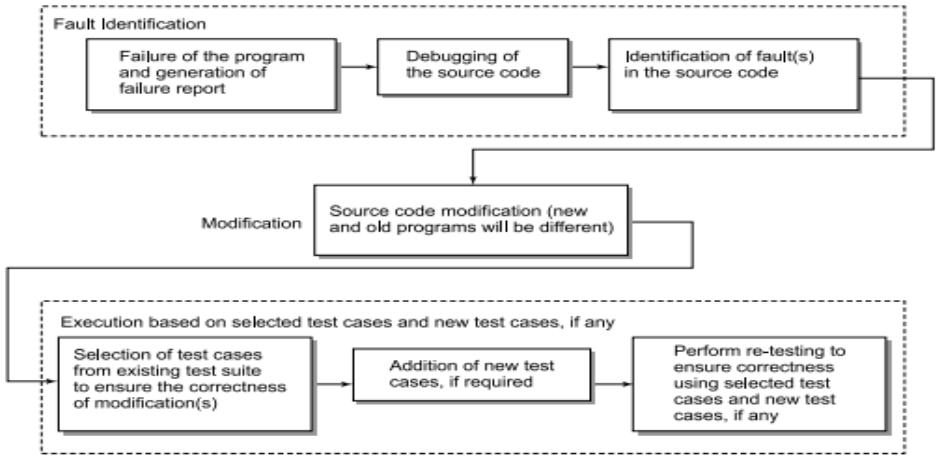
(c) S, N1, N2, N7, N8, N9, N11, N12, D

(d) S, N1, N2, N8, N10, N11, N12, D

8.7 PRACTICAL 7- SELECTION, MINIMIZATION AND PRIORITIZATION OF TEST CASES FOR REGRESSION TESTING

Regression testing is about re-executing all the test cases every time a change is incorporated to an application. This is to ensure if any change in one module has not affected the other part of the module. It is a very costly process and consumes a significant amount of resources. The challenge is on finding out ways to reduce the cost. Whenever a failure is experienced, it is reported to the software team. The team may like to debug the source code to know the reason(s) for this failure. After identification of the reason(s), the source code is modified and we generally do not expect the same failure again. In order to ensure this correctness, we re-test the source code with a focus on modified portion(s) of the source code and also on affected portion(s) of the source code due to modifications. We need test cases that target the modified and affected portions of the source code. We may write new test cases, which may be a 'time and effort consuming' activity. We neither have enough time nor reasonable resources to write new test cases for every failure. Another option is to use the existing test cases which were designed for development testing and some of them might have been used during development testing. The existing test suite may be useful and may reduce the cost of regression testing. As we all know, the size of the existing test suite may be very large and it may not be possible to execute all tests. The greatest challenge is to reduce the size of the existing test suite for a particular failure. The various steps are shown in the figure given below. Hence, **test case selection** for a failure is the main key for regression testing.

Steps of Regression Testing Process:



8.7.1 Practical 7a : Test case minimization in Regression Testing:

A Web service is a program that can be used by another program over the Web. Consider a Web service named ZC, short for ZipCode. The initial version of ZC provides two services: ZtoC and ZtoA. Service ZtoC inputs a zip code and returns a list of cities and the corresponding state while

ZtoA inputs a zip code and returns the corresponding area code. We assume that while the ZipCode service can be used over the Web from wherever an internet connection is available, it serves only the United States. Let us suppose that ZC has been modified to ZC' as follows. First, a user can select from a list of countries and supply the zip code to obtain the corresponding city in that country. This modification is made only to the ZtoC function while ZtoA remains unchanged. Note that the term "zip code" is not universal. For example, in India, the equivalent term is "pin code" which is 6-digits long as compared to the 5-digit zip code used in the United States. Second, a new service named ZtoT has been added which inputs a country and a zip code and returns the corresponding time zone.

Consider the following two tests (only inputs specified) used for testing ZC.

t1 : < service = ZtoC , zip = 47906 >

t2 : < service = ZtoA, zip = 47906 >

A simple examination of the two tests reveals that test t1 is not valid for ZC' as it does not list the required country field. Test t2 is valid as we have made no change to ZtoA.

Thus we need to either discard t1 and replace it by a new test for the modified ZtoC, or simply modify t1 appropriately. We prefer to modify and hence our validated regression test suite for ZC' is

t1 : < country = USA, service = ZtoC , zip = 47906 >

t2 : < service = ZtoA, zip = 47906 > .

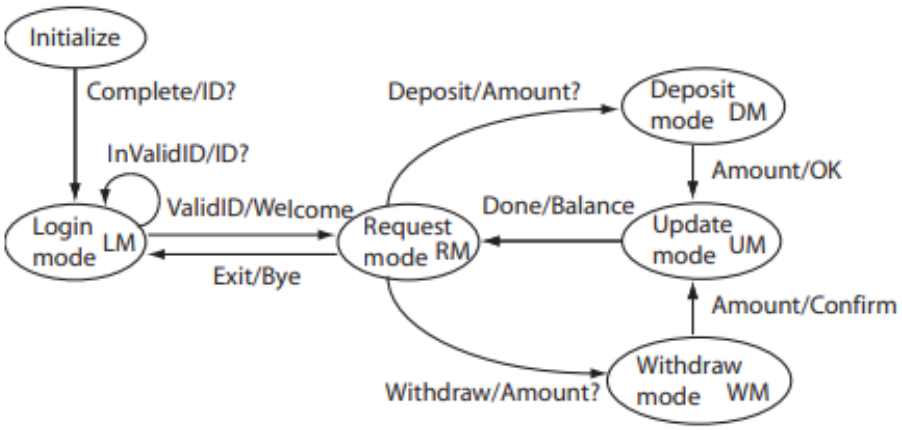
Note that testing ZC' requires additional tests to test the ZtoT service. However, we need only the two tests listed above for regression testing.

To keep this example short, we have listed only a few tests for ZC. In practice one would develop a much

larger suite of tests for ZC which will then be the source of regression tests for ZC'.

8.7.2 Practical 7b- Test Sequencing in regression testing:

Consider a simplified banking application referred to as SATM. Application SATM maintains account balances and offers users the following functionality: login, deposit, withdraw, and exit. Data for each account is maintained in a secure database.



In the above figure State transition in a simplified banking application. Transitions are labeled as X/Y, where X indicates an input and Y the expected output. "Complete" is an internal input indicating that the application moves to the next state upon completion of operations in its current state.

Figure above exhibits the behavior of SATM as a finite state machine. Note that the machine has six distinct states, some referred to as modes in the figure. These are labeled as Initialize, LM, RM, DM, UM, and WM. When launched, the SATM performs initialization operations, generates an "ID?" message, and moves to the LM state. If the user enters a valid ID, SATM moves to the RM state else it remains in the LM state and again requests for an ID. While in the RM state the application expects a service request. Upon receiving a Deposit request it enters the DM state and asks for an amount to be deposited. Upon receiving an amount it generates a confirmatory message and moves to the UM state where it updates the account balance and gets back to the RM state. A similar behavior is shown for the Withdraw request. SATM exits the RM state upon receiving an Exit request. Let us now consider a set of three tests designed to test the Login, Deposit, Withdraw and Exit features of SATM. The tests are given in the following table in the form of a test matrix. Each test requires that the application be launched fresh and the user (tester in this case) log in. We assume that the user with ID=1 begins with an account balance of 0. Test t1 checks the login module and the Exit feature, t2 the Deposit module, and t3 the Withdraw module. As you might have guessed, these tests are not sufficient for a thorough test of SATM, but they suffice to illustrate the need for test sequencing as explained next.

Test	Input sequence	Expected output sequence	Purpose
t_1	ID=1, Request= Exit	Welcome, Bye	Test Login module.
t_2	ID=1, Request= Deposit, Amount=50	D?, Welcome, Amount?, OK, Done, 50	Test Deposit module.
t_3	ID=1, Request= Withdraw, Amount=30	ID?, Welcome, Amount?, 30, Done, 20	Test Withdraw module.

Now suppose that the Withdraw module has been modified to implement a change in withdrawal policy, e.g. "No more than $300 can be withdrawn on any single day." We now have the modified SATM' to be tested for the new functionality as well as to check if none of the existing functionality has broken. What tests should be rerun? Assuming that no other module of SATM has been modified, one might propose that tests t1 and t2 need not be rerun. This is a risky proposition unless some formal technique is used to prove that indeed the changes made to the Withdraw module cannot affect the behavior of the remaining modules. Let us assume that the testers are convinced that the changes in SATM will not affect any module other than Withdraw. Does this mean that we can run only t3 as a regression test ? The answer is in the negative. Recall our assumption that testing of SATM begins with an account balance of 0 for the user with ID=1. Under this assumption, when run as the first test, t3 will likely fail because the expected output will not match the output generated by SATM' (see Exercise 11.1). The argument above leads us to conclude that we need to run test t3 after having run t2. Running t2 ensures that SATM' is brought to the state in which we expect test t3 to be successful. Note that the finite state machine shown in Figure 11.3 ignores the values of internal variables and data bases used by SATM and SATM'. During regression as well as many other types of testing, test sequencing is often necessary to bring the application to a state where the values of internal variables, and contents of the data bases used, correspond to the intention at the time of designing the tests. It is advisable that such intentions (or assumptions) be documented along with each test.

8.7.3 Practical 7c: Selection of Test Cases in Regression Testing:

We want to use the existing test suite for regression testing. How should we select an appropriate number of test cases for a failure? The range is from "one test case" to "all test cases". A 'regression test cases' selection technique may help us to do this selection process. The effectiveness of the selection technique may decide the selection of the most appropriate test cases from the test suite. Many techniques have been developed for procedural and object oriented programming languages. Testing professionals are, however, reluctant to omit any test case from a test suite that might expose a fault in the modified program. We consider a program given in Figure below along with its modified version where the modification is in line 6 (replacing operator '*' by '-'). A test suite is also given in Table given below.

```
1.  main( )                          1.  main ( )
2.  {                                2.  {
3.  int a, b, x, y, z;               3.  int a, b, x, y, z;
4.  scanf ("%d, %d", &a, &b);        4.  scanf ("%d, %d", &a, &b);
5.  x = a + b ;                      5.  x = a + b;
6.  y = a* b;                        6.  y = a – b;
7.  if (x ≥ y) {                     7.  if (x ≥ y) {
8.  z = x / y ;                      8.  z = x / y ;
9.  }                                9.  }
10. else {                           10. else {
11. z = x * y ;                      11. z = x * y ;
12. }                                12. }
13. printf ("z = %d \ n", z );       13. printf ("z = %d \ n", z);
14. }                                14. }
```

(a) Original program with fault in line 6. (b) Modified program with modification in line 6.

Test Cases:

Set of Test Cases

S. No.	Inputs		Execution History
	a	b	
1	2	1	1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 14
2	1	1	1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 14
3	3	2	1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14
4	3	3	1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14

In this case, the modified line is line number 6 where 'a*b' is replaced by 'a-b'. All four test cases of the test suite execute this modified line 6. We may decide to execute all four tests for the modified program. If we do so, test case 2 with inputs a = 1 and b = 1 will experience a 'divide by zero' problem, whereas others will not. However, we may like to reduce the number of test cases for the modified program. We may select all test cases which are executing the modified line. Here, line number 6 is modified. All four test cases are executing the modified line (line number 6) and hence are selected. There is no reduction in terms of the number of test cases. If we see the execution history, we find that test case 1 and test case 2 have the same execution history. Similarly, test case 3 and test case 4 have the same execution history. We choose any one test case of the same execution history to avoid repetition. For execution history 1 (i.e. 1, 2, 3, 4, 5, 6, 7, 8, 10, 11), if we select test case 1, the program will execute well, but if we select test case 2, the program will experience 'divide by zero' problem. If several test cases execute a particular modified line, and all of these test cases reach a particular affected source code segment, minimization methods require selection of only one such test case, unless they select the others for coverage elsewhere. Therefore, either test case 1 or test case 2 may have to be selected. If we select test case 1, we miss the opportunity to detect the fault that test case 2 detects. Minimization techniques may omit some test cases that might expose fault(s) in the modified program. Hence, we should be very careful in the process of minimization of test cases and always try to use safe regression test selection technique (if at all it is possible). A safe regression test selection technique should select all test cases that can expose faults in the modified program

8.7.4 Practical 7 d: Test Case Prioritization in Regression Testing

Many times, on account of time constraints it is difficult to test all the test cases again so we may prioritize the test cases so that you may first execute the top most priority test cases and then the others to ensure important test cases are not missed upon.

The following are the Test Scenarios of an application under testing

Test Scenario ID	Reference	Test Scenario Description	Number of Test Cases
(TS_001) Register Functionality	Application URL	Verify the working of Register Account functionality	27
(TS_002) Login Functionality	Application URL	Verify the working of Login functionality	23
(TS_003) Logout Functionality	Application URL	Verify the working of Logout functionality	11
(TS_004) Forgot Passsword	Application URL	Verify the working of Forgot Password functionality	25
(TS_005) Search Functionality	Application URL	Verify the working of Search functionality	22
(TS_006) Product Compare	Application URL	Verify the working of Product Compare functionality	24
(TS_007) Product Display Page	Application URL	Verify the Product Display Page functionality for the different types of Products	37
(TS_008) Add to Cart	Application URL	Verify the working of 'Add to Cart' functionality	09
(TS_009) Wish List	Application URL	Verify the working of 'Wish List' functionality	21
(TS_017) Order History	Application URL	Verify the working of My Orders > 'Order History' functionality	12
(TS_018) Order Information	Application URL	Verify the working of My Orders > 'Order Information' functionality	8
(TS_019) Product Returns	Application URL	Verify the working of My Orders > 'Product Returns' functionality	11

(TS_020) Downloads	Application URL	Verify the working of My Orders > 'Downloads' functionality	13
(TS_021) Reward Points	Application URL	Verify the working of My Orders > 'Reward Points' functionality	10
(TS_022) Returns Page	Application URL	Verify the working of My Orders > 'Returned Requests' functionality	17
(TS_023) Transactions	Application URL	Verify the working of My Orders > 'Your Transactions' functionality	11

Each of the test scenarios have some test cases for example the first one Registration Functionality has 27 test cases. So we need to prioritize them. Lets consider the Test scenario Registration Functionality test cases for minimization:

Will consider three priority P1, P2 and P3 where P1 has High Importance, P2 is Moderately Important and P3 importance is low.

Test Case ID	Test Scenario	Test Case Title	Pre-requisites	Test Steps	Test Data	Expected Result (ER)	Actual Result	Priority	Result	Comments
<< Test Scenarios										
TC_RF_001	(TS_001) Register Functionality	Verify Registering an Account by providing only the Mandatory fields	1. Open the Application (http://tutorialsninja.com/demo) in any Browser	1. Click on 'My Account' Drop menu 2. Click on 'Register' option 3. Enter new Account Details into the Mandatory Fields (First Name, Last Name, E-Mail,Telephone, Password, Password Confirm and Privacy Policy Fields) 4. Click on 'Continue' button (ER-1) 5. Click on 'Continue' button that is displayed in the 'Account Success' page (ER-2)	Not Applicable	1. User should be logged in, taken to 'Account Succcess' page and proper details should be displayed on the page 2. User should be taken to 'Account' page and a confirm email should be sent to the registed email address		P1		
TC_RF_002	(TS_001) Register Functionality	Verify 'Thank you for registering' email is sent to the registered email address as a confirmation for registering the account	1. Open the Application (http://tutorialsninja.com/demo) in any Browser	1. Click on 'My Account' Drop menu 2. Click on 'Register' option 3. Enter new Account Details into the Mandatory Fields (First Name, Last Name, E-Mail,Telephone, Password, Password Confirm and Privacy Policy Fields) 4. Click on 'Continue' button 5. Check the email address used for registering the account (Verify ER-1, ER-2. ER-3) 6. Click on the Login page link from the Email body (Verify ER-4)	Not Applicable	1. Verify a confirmation email for registering the account is sent to the registered email address. 2. Verify the Email subject, body and from address of the received email. 3. Verify there is a link to the login page provided in the Email body		P2		

						4. User should be taken to the Login page				
TC_RF_003	(TS_001) Register Functionality	Verify Registering an Account by providing all the fields	1. Open the Application (http://tutorialsninja.com/demo) in any Browser	1. Click on 'My Account' Drop menu 2. Click on 'Register' option 3. Enter new Account Details into all the Fields (First Name, Last Name, E-Mail,Telephone, Password, Password Confirm, Newsletter and Privacy Policy Fields) 4. Click on 'Continue' button (ER-1) 5. Click on 'Continue' button that is displayed in the 'Account Success' page (ER-2)	Not Applicable	1. User should be logged in, taken to 'Account Succcess' page and proper details should be displayed on the page 2. User should be taken to 'Account' page		P2		
TC_RF_004	(TS_001) Register Functionality	Verify proper notification messages are displayed for the mandatory fields, when you don't provide any fields in the 'Register Account' page and submit	1. Open the Application (http://tutorialsninja.com/demo) in any Browser	1. Click on 'My Account' Drop menu 2. Click on 'Register' option 3. Don't enter anything into the fields 4. Click on 'Continue' button (ER-1)	Not Applicable	1. The below warning messages should be displayed for the respective fields: For First Name field, the warning message 'First Name must be between 1 and 32 characters!' should be displayed For Last Name field, the warning message 'Last Name must be between 1 and 32 characters!' should be displayed For E-Mail field, the warning message 'E-Mail Address does not appear to be valid!' should be displayed For Telephone field, the warning message 'Telephone must be between 3 and 32 characters!' should be displayed For Password field, the warning message 'Password must be between 4 and 20 characters!' should be displayed For Privacy Policy field, the warning message 'Warning: You must agree to the Privacy Policy!' should be		P3		

						displayed on the top				
TC_RF_005	(TS_001) Register Functionality	Verify Registering an Account when 'Yes' option is selected for Newsletter field	1. Open the Application (http://tutorialsninja.com/demo) in any Browser	1. Click on 'My Account' Drop menu 2. Click on 'Register' option 3. Enter new Account Details into all the Fields (First Name, Last Name, E-Mail,Telephone, Password, Password Confirm and Privacy Policy Fields) 4.Click on 'Yes' radio option for Newsletter 5. Click on 'Continue' button (ER-1) 6. Click on 'Continue' button that is displayed in the 'Account Success' page (ER-2) 7. Click on 'Subscribe/unsubscribe to newsletter' option (ER-3)	Not Applicable	1. User should be logged in, taken to 'Account Succcess' page and proper details should be displayed on the page 2. User should be taken to 'Account' page 3. 'Yes' option should be displayed as selected by default in the Newsletter page		P3		

The first test case is set as Priority P1 as it covers all mandatory field without which registration cannot happen. The second and third test case has the priority P2 as it is important but not mandatory the error on account of that can be tolerable.

The fourth and fifth test case is given priority P3 as its most coverage already happens in the first two test cases.

8.8 PRACTICAL 8- VALIDATION TESTING

Guidelines for Validity Checks:

1. Data Type If input x is defined as an integer, then x should also be checked for float, char, double, etc. values. We should clearly state what can be accepted as an input. In the login form, (please refer below, we should clearly state the type of both the inputs i.e. Login Id and password. For example, the Login Id input should be numeric and should not accept alphabets, special characters and blank spaces. Similarly, the password input will accept alphabets, digits, hyphen and underscore but will not accept blank spaces. We should generate validity checks for every ‘do’ and every ‘do not’ case.

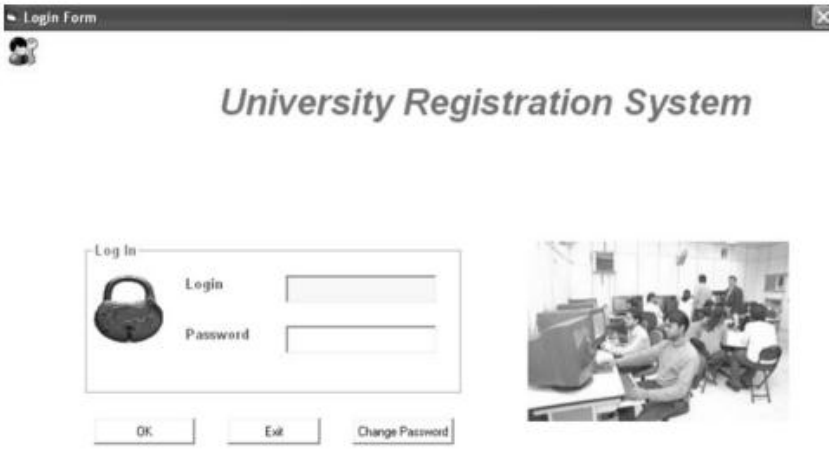
2. Data Range The range of inputs should also be clearly specified. If x is defined as an integer, its range, (say 1 x 100) should also be defined. Validity checks may be written for conditions when x 1 and x > 100. For example, in login form, length of the login-id is defined as 11 digits and the password as 4 to 15 digits. We should generate validity checks for both valid and invalid range of inputs.

3. Special Data Conditions Some special conditions may need to be checked for specified inputs. For example, in the e-mail address, ‘@’ and ‘.’symbols are essential and must be checked. We should write validity checks for such special symbols which are essential for any specific input.

4. Mandatory Data Inputs Some inputs are compulsory for the execution of a program. These mandatory fields should be identified and validity checks be written accordingly. In the login form, both inputs (login Id and password) are mandatory. Some fields (data inputs) may not be mandatory like telephone number in a student registration form. We should provide validity checks to verify that mandatory fields are entered by the user.

5. Domain Specific Checks Some validity checks should be written on the basis of the expected functionality. In the URS, no two semesters should have a common paper. The roll number should be used as a Creating Test Cases from Requirements and Use Cases 317 login Id. A student cannot select more than the required number of elective papers in a semester. These domain specific issues should be written as validity checks in order to verify their correctness.

8.8.1 Practical 8a : Validation Testing on Login Functionality:



Validity Checks for Login Form:

Validity check Number	Description
VC1	Every user will have a unique login Id.
VC2	Login Id cannot be blank.
VC3	Login Id can only have 11 digits.
VC4	Login Id will not accept alphabetic, special and blank spaces.
VC5	Password cannot be blank.
VC6	Length of password can only be 4 to 15 digits.
VC7	Alphabets, digits, hyphen and underscore characters are allowed in password field.
VC8	Password will not accept blank spaces.

Test Cases:

Test case Id	Validity check Number	Login id	Password	Expected output	Remarks
TC1	VC1	10234567899	Rkhj7689	User successfully logs into the system	-
TC2	VC2		*	Please Enter Login Id	Login id cannot be blank
TC3	VC3	1234	*	Invalid login id	Login id should have 11 digits
TC4	VC4	Ae455678521	*	Invalid login id	Login id cannot have alphanu-meric characters
TC5	VC4	123$4567867	*	Invalid login id	Login id cannot have special characters
TC6	VC4	123 45667897	*	Invalid login id	Login id cannot have blank spaces
TC7	VC5	10234567899		Please Enter Password	Password cannot be blank
TC8	VC6	10234567899	Ruc	Invalid password	Password cannot be less than 4 characters in length

8.8.2 Practical 8b:

Validity Testing for Change Password Form:

Validity Checks for Change Password Form:

Validity check Number	Description
VC9	Login Id cannot be blank.
VC10	Login Id can only have 11 digits.
VC11	Login Id will not accept alphabetic, special and blank spaces.
VC12	Old password cannot be blank.
VC13	Length of old password can only be 4 to 15 digits.
VC14	Alphabets, digits, hyphen and underscore characters are allowed in old password field.
VC15	Old password will not accept blank spaces.
VC16	New password cannot be blank.
VC17	Length of new password can only be 4 to 15 digits.
VC18	Alphabets, digits, hyphen and underscore characters are allowed in new password field.
VC19	New password will not accept blank spaces.
VC20	'Confirm password' cannot be blank.
VC21	'Confirm password' should match with new password.

Test Cases with Actual Data:

Test case Id	Validity check No.	Login id	Old password	New Password	Confirm Password	Expected output	Remarks
TC1	VC9		*	*	*	Please Enter Login Id	Login id cannot be blank
TC2	VC10	1234	*	*	*	Invalid login id	Login id should have 11 digits
TC3	VC11	Ae455678521	*	*	*	Invalid login id	Login id cannot have alphanumeric characters
TC4	VC11	123$4567867	*	*	*	Invalid login id	Login id cannot have special characters
TC5	VC11	123 45667897	*	*	*	Invalid login id	Login id cannot have blank spaces
TC6	VC12	10234567899		*	*	Please Enter Old Password	Password cannot be blank
TC7	VC13	10234567899	Ruc	*	*	Invalid old password	Password cannot be less than 4 characters long
TC8	VC14	10234567899	Rtyuiopki1123678	*	*	Invalid old password	Password cannot be greater than 15 characters in length
TC9	VC14	10234567899	Rty_uyo	*	*	--	Password can have underscore character
TC10	VC15	10234567899	Rt yuii	*	*	Invalid old password	Password cannot have blank spaces
TC11	VC16	10234567899	Ruc_ui		*	--	Password cannot be blank
TC12	VC17	10234567899	Ruc_ui	Rrk	*	Invalid new password	Password cannot be less than 4 characters long
TC13	VC17	10234567899	Ruc_ui	Rtyuiopki1123678	*	Invalid new password	Password cannot be greater than 15 characters in length
TC14	VC18	10234567899	Ruc_ui	Rty_uyo	*	Invalid new password	New password can have underscore character
TC15	VC19	10234567899	Ruc_ui	Rty uyo	*	Invalid new password	New password cannot have blank spaces
TC16	VC20	10234567899	Ruc_ui	Rty_uyo		--	'Confirm Password' cannot be blank
TC17	VC21	10234567899	Ruc_ui	Rty_uyo	Rty_uyo	Invalid 'Confirm Password'	'Confirm Password' should match with new password

8.9 Practical 9- Adhoc Testing

Ad-hoc Testing:

Ad-hoc testing is an informal and improvisational approach to assessing the viability of a product. An ad-hoc test is usually only conducted once unless a defect is found. Commonly used in software development, ad hoc testing is performed without a plan of action and any actions taken are not typically documented. Testers may not have detailed knowledge of product requirements. Ad hoc testing is also referred to as random testing and monkey testing.

Because the approach is non-methodical, ad hoc testing can miss flaws that would be found in a more structured testing system. However, the lack of formal requirements also means that obvious flaws can be attended to more quickly than if they had to be approached in a more systematic fashion.

8.9.1- Practical 9a:

Consider an automated banking application. The user can dial the bank from a personal computer, provide a six-digit password, and follow with a series of keyword commands that activate the banking function. The software for the application accepts data in the following form:

Area Code	Blank or three-digit number
Prefix	Three-digit number, not beginning with 0 or 1
Suffix	Four-digit number
Password	Six-character alphanumeric
Commands	"Check status", "Deposit", "Withdrawal"

Design ad-hoc test cases to test the system.

Test Case ID	Test Step	Description	Test Data	Expected result	Actual result	Status (Pass/ Fail)
TC01	validation of area code	Enter the 3 digit area code if non- local and Blank if local.				
TC02	validation of area code	enter the 3 digit area code if non- local and Blank if local.				
TC03	validation of area code	enter the 3 digit area code if non- local and Blank if local.				
TC04	validation of area code	enter the 3 digit area code if non- local and Blank if local.				
TC05	Validation of Prefix	enter a 3-digit number prefix. It should not begin with 0 (or) 1.				
TC06	Validation of Prefix	enter a 3-digit number prefix.It should not begin with 0 (or) 1.				

*****