Experiment 9: Apply the knowledge of test cases for the project using white box testing.

Learning Objective: Students will able to create unit test cases

Tools: VsCode

Theory:

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation.

Unit Testing:

Unit testing focuses on the building blocks of the software system, that is, objects and subsystems. The specific candidates for unit testing are chosen from the object model and the system decomposition. In principle, all the objects developed during the development process should be tested, which is often not feasible because of time and budget constraints. The minimal set of objects to be tested should be the participating objects in the use cases. Subsystems should be tested after each of the objects and classes within that subsystem have been tested individually Unit testing focuses verification effort on the smallest unit of software design—the software component or module. The unit test is white-box oriented. . In Unit testing the following are tested,

1. The module interface is tested to ensure that information properly flows into and out of the program unit under test.
2. The local data structure is examined to ensure that data stored temporarily maintains its integrity.
3. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
4. All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
5. And finally, all error handling paths are tested

Write a program to calculate the square root of a number in the range 1-100

```
def squareroot(n):
    sq = n**0.5
```

```
    print("Square root of "+str(n)+ " is "+ str(sq))


n = int(input("Enter a number to find square: "))
if(0<n<50):
    if(n>0):
        if(isinstance(n, int)):
            s = squareroot(n)
        else:
            print("Enter Integer number")
elif (n<0):
    print("Negative number")
else:
    print("Beyond the Range")
```

Sr no	Input	Output
1	-2	Beyond the range
2	0	Beyond the range
3	1	Square root of 1 is 1
4	100	Square root of 100 is 10
5	102	Beyond the range
6	-16	Negative number
7	62	Square root of 62 is 7.874

Test Cases

Test case 1 : {I1 ,O1}
Test case 2 : {I2 ,O2}
Test case 3 : {I3, O3}
Test case 4 : {I4, O4}
Test case 5 : {I5, O5}
Test case 6 : {I6, O6}
Test case 7 : {I7, O7}

Selenium:

Selenium IDE is **an integrated development environment for Selenium tests**. It is implemented as a Firefox extension, and allows you to record, edit, and debug tests. Selenium is an open-source tool that automates web browsers. It provides a single interface that lets you write test scripts in programming languages like Ruby, Java, NodeJS, PHP, Perl, Python, and C#, among others.

A browser-driver then executes these scripts on a browser-instance on your device (more on this in a moment).

History of Selenium:

A timeline of major events in the evolution of Selenium from an in-house side-project to an open-source industry standard in browser automation:

2004: Making history in two parts (from Selenium A to B)

- Jason Huggins of ThoughtWorks needs to test his web app's front-end behavior across different browsers.
- He develops a tool that works by injecting JavaScript underneath the webpage, allowing the tester to write code that could 'automate' front-end user interactions. This became the JavaScript TestRunner.
- Although the JS-injection approach couldn't naturally replicate user interactions (via keystrokes/mouse movements), it was a workaround for the 'same-host origin policy', which prohibits external JavaScript code from accessing elements from a domain it didn't originally reside in. Nonetheless, the tool is positively received by in-house developers and ThoughtWorks' clients alike.
- **The tool is open sourced due to popular demand.**
- To eliminate the need for JS-injections, Huggins, along with colleague Paul Hammant, discuss the possibility of a 'server' component. This server would act as an HTTP proxy and trick the browser-instance into believing that the test script and the web app under test are from the same source.
- They develop the server component in Java and the original client-side driver (TestRunner) gets ported to Ruby.
- **This is the original Selenium.** Known as Driven Selenium or Selenium B in the evolution timeline.

2005: Selenium RC (Remote Control)

- Elsewhere (at Bea, specifically), Dan Fabulich and Nelson Sproul begin working on the driver coder. They eventually mold it into a standalone server that bundled MortBay's Jetty as HTTP proxy.
- **This becomes 'Selenium RC (Remote Control)' or Selenium 1.0.** Before we cut to 2.0, there is another significant development in the form of…

2006: The Selenium IDE

- Shinya Kasatani wraps the Selenium driver code in an IDE module in Firefox browser.
- When it works, he finds that he can run a functional 'live test' on a website—interacting with the browser (as a user would); recording/replaying the interactions and debugging as needed.
- Kasatani donates this tool to Selenium project where **it becomes known as the Selenium IDE.**

2007: The Selenium WebDriver (Selenium 2.0)

- Back at ThoughtWorks, Simon Stewart diligently codes up separate 'driver' clients for every popular browser, so they'd all support automation with native browser capabilities.
- It pays off. **The project becomes famous as the WebDriver.**

2008: Multiply by 'n': The Selenium Grid

- At ThoughtWorks, Philippe Hanrigou creates a server which would allow testers to access and run tests on browser instances on any number of remote devices.
- **This becomes known as the Grid.** Cut to…

2016: Selenium RC gets deprecated and WebDriver becomes standard implementation—aka **Selenium 3.0.**

2019: WebDriver becomes a W3C standard protocol

Imagine that a manual tester has this scenario: Checking whether the web app's signup page (www.example.com/signup) validates input strings and registers a user successfully in latest versions of Chrome and Firefox, on Windows 7.

Assume that the signup page has these input fields—username, email address, and password. The tester will get a Windows 7 desktop and follow these steps, consecutively, on latest versions of Chrome and Firefox:

1. Enter the URL in the address bar (www.example.com/signup)
2. Enter an invalid string in each input field (email, username, and password)
3. Check whether the input strings were validated against corresponding regexes and any pre-existing values in the database
4. Enter 'valid' strings in each input field; click Sign Up
5. Check whether "Welcome, '{'username'}'" page showed up
6. Check whether the system database created a new userID for '{'username'}'
7. Mark the test 'passed' if it did, 'failed' if the signup feature broke anywhere during the test.

Types of testing that are commonly automated with Selenium are:

1. **Compatibility Testing:**
   Done by QA professionals/Testers to ensure that the web app meets performance benchmarks on different browser-OS combinations. For example, testing on different devices (mobile and desktop) to ensure that the front-end fits to scale (responsive); testing on different browsers to see if video ads render on the pages as they should.

2. **Performance Testing:**
   Series of tests done by QA professionals/Testers to ensure that the project meets performance benchmarks set by the stakeholders. Tester writes a script that checks whether all elements on homepage load within 2 seconds on different browsers/browser versions.

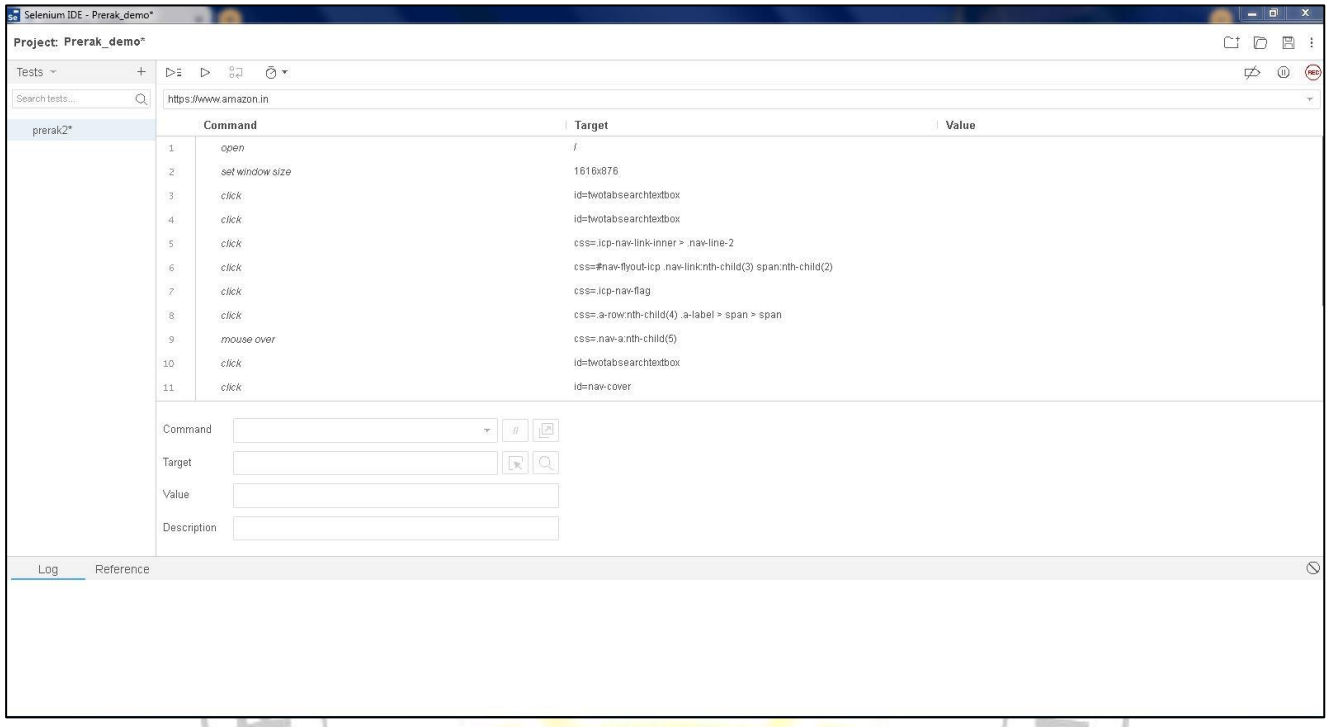
3. **Integration Testing:**
   Done by developers to verify that units/modules coded separately (that work on their own), also work when put together. Parallel Test Calculator, for instance, has separate layers. UI takes input and business logic calculates the output—then sends it back to UI to display. The tester could verify whether they are able to relay data/output when integrated.

4. **System Testing:**
   aka Black Box testing. Done by Testers/QA professionals with no context of the code or any previously executed tests. Typically centered on a single user workflow. The check-out process on a product website, for instance, comprises of: validating user credentials, fetching products from the cart, checking their availability, and validating payment details—before redirecting to the bank website. The tester could write a script to verify that the entire system is functional.

5. **End-to-end Testing:**
   Also done by Testers/QA professionals, typically from the user's point of view. The aim is to verify that all touchpoints on the web app are functional. From the previous example, the tester could write a series of test cases to check that sign-up, product search, checkout, review, bookmark, and all other features function as intended (and fail when invalid values are entered in input fields).

6. **Regression Testing:**
   A series of tests done to ensure that newly built features work with the existing system. From the same example, say the product website launches a new feature (promotional codes) that automatically apply to eligible items before checkout. The tester could write cases to verify that it doesn't break the rest of the checkout feature.
   Well-written test suites can also automate Smoke and Sanity testing with Selenium.

Implementation:



Learning Outcomes: Students should have the ability to

LO1: Students will be able to understand Software Testing Concepts and the various Software standards.
LO2: to test a software with the help of Junit
LO3: create test cases
LO4: To understand different tools for testing

Outcomes: Upon completion of the course students will be able to write test cases for the project.

Conclusion:

Successfully understood and implemented Software Testing Concepts and the various Software standards and unit testing.

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	
Marks Obtained				