

Practical No. 11 Implementing Basic Automation Frameworks

Date: _____

Aim:

To study Basic Automation Frameworks - linear scripting, library architecture framework, data driven Framework.

Theory:

Automation frameworks are structured methods for developing, organizing, and executing automated tests. They help streamline the testing process, making it more efficient and maintainable. Here's a detailed look at three common types of automation frameworks: linear scripting, library architecture, and data-driven frameworks.

1. Linear Scripting Framework

Description:

The linear scripting framework is the simplest form of test automation. In this approach, test scripts are written as a sequence of actions, directly reflecting the steps a user would take in an application.

Features:

- **Sequential Execution:** Each line of code corresponds to a specific action or step in the test case.
- **Easy to Understand:** The linear format makes it straightforward for testers and stakeholders to follow along.

Advantages:

- **Simplicity:** Ideal for small projects where test cases are straightforward and few.
- **Quick Setup:** Easy to write and implement without extensive planning.

Disadvantages:

- **Poor Maintainability:** As test cases grow, maintaining them can become cumbersome; any change requires updating multiple scripts.
- **Limited Reusability:** Code is often duplicated across scripts, leading to inconsistencies and errors.
- **Scalability Issues:** Difficult to manage as the number of tests increases.

2. Library Architecture Framework

Description:

The library architecture framework improves upon the linear scripting approach by organizing reusable code into libraries. This structure allows common functions or actions to be reused across multiple test cases.

Features:

- **Modular Design:** Test logic is separated into libraries (e.g., functions for logging in, navigating, etc.).
- **Centralized Code:** Common functionalities are maintained in one place, which can be updated as needed.

Advantages:

- **Reusability:** Code can be reused across different tests, reducing redundancy.
- **Improved Maintenance:** Changes to a function need to be made only once in the library rather than in multiple scripts.
- **Better Organization:** Promotes a cleaner structure, making it easier to manage and understand.

Disadvantages:

- **Initial Setup:** Requires more effort upfront to create the libraries and organize the structure.
- **Complexity:** If not well-structured, it can lead to confusion about where to find specific functions or how to use them.

3. Data-Driven Framework**Description:**

The data-driven framework separates test logic from test data. It allows the same test script to be executed with multiple sets of data, enabling comprehensive testing of various scenarios.

Features:

- **Separation of Data and Logic:** Test scripts reference external data sources (like spreadsheets or databases) instead of hardcoding data.
- **Flexible Test Execution:** The same test logic can be executed with different input values, which is particularly useful for validating the same functionality under various conditions.

Advantages:

- **High Reusability:** Tests can be reused with different data sets without needing to duplicate test logic.
- **Efficient Management of Scenarios:** Easy to manage large sets of test data; adding new

scenarios often only requires adding new data rather than writing new scripts.

- **Better Coverage:** Facilitates more thorough testing by easily running the same test with different inputs.

Disadvantages:

- **Setup Complexity:** Requires a more sophisticated framework to manage the separation of data and test logic.
- **Data Management Overhead:** Test data needs to be carefully managed to ensure it is accurate and up-to-date.

Parameterization of test

One of the important features of TestNG is parameterization.

This feature allows users to pass parameter values to test methods as arguments. This is supported by using the Parameters and DataProvider annotations.

There are mainly two ways through which we can provide parameter values to test-methods:

- Through TestNg XML configuration file
- Through DataProviders

Parameterization through testng.xml

If you need to pass some simple values such as String types to the test methods at runtime, you can use this approach of sending parameter values through TestNG XML configuration files.

You have to use Parameters annotation for passing parameter values to the test method.

DataProvider

One of the important features provided by TestNG is the DataProvider feature.

It helps the user to write data-driven tests, that means same test method can be run multiple times with different datasets.

DataProvider is the second way of passing parameters to test methods. It helps in providing complex parameters to the test methods as it is not possible to do this from XML.

To use the DataProvider feature in your tests you have to declare a method annotated by @DataProvider and then use the said method in the test method using the dataProvider attribute in the Test annotation.

Implementation

1. Write a test class containing test method that calculates the average marks that awarded by two reviewers prints whether writer is shortlisted if average is >4. The marks are passed as parameters whose values are passed from testing.xml at test level.

Code :

```
package prac11;

import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

public class Problem1 {

    @Parameters({"mark1", "mark2"})
    @Test
    public void calculateAverageMarks(String mark1, String mark2) {
        // Parse the string marks to integers
        int marks1 = Integer.parseInt(mark1);
        int marks2 = Integer.parseInt(mark2);

        // Calculate the average
        double average = (marks1 + marks2) / 2.0;

        // Print the average marks
        System.out.println("Average Marks: " + average);

        // Check if the writer is shortlisted
        if (average > 4) {
            System.out.println("The writer is shortlisted.");
        } else {
            System.out.println("The writer is not shortlisted.");
        }
    }
}
```

```
}
```

XML File :

```
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
```

```
<suite name="Problem1">
```

```
  <test name="Problem1">
```

```
    <parameter name="mark1" value="5"/>
```

```
    <parameter name="mark2" value="3"/>
```

```
    <classes>
```

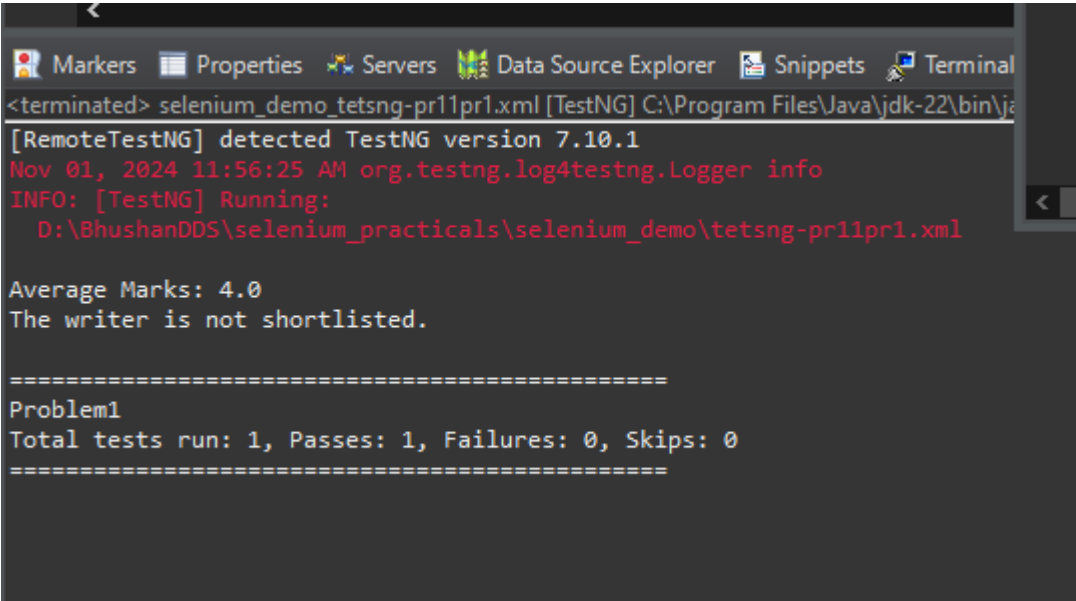
```
      <class name="prac11.Problem1"/>
```

```
    </classes>
```

```
  </test>
```

```
</suite>
```

Output :



The screenshot shows a terminal window with the following output:

```
<terminated> selenium_demo_tetsng-pr11pr1.xml [TestNG] C:\Program Files\Java\jdk-22\bin\ja
[RemoteTestNG] detected TestNG version 7.10.1
Nov 01, 2024 11:56:25 AM org.testng.log4testng.Logger info
INFO: [TestNG] Running:
      D:\BhushanDDS\selenium_practicals\selenium_demo\tetsng-pr11pr1.xml

Average Marks: 4.0
The writer is not shortlisted.

=====
Problem1
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
=====
```

2. Implement a test class named LoginTest. Create a method that provides test data (username and password) using the `@DataProvider` annotation. Create a test method named `testLogin` that takes two parameters (username and password). If the username and password are both valid, the test should assert a successful login message else display an error message indicating the login failure.

Code :

```
package prac11;

import org.testng.Assert;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class Problem2 {

    // DataProvider method to supply test data for login
    @DataProvider(name = "loginData")
    public Object[][] provideLoginData() {
        return new Object[][] {
            {"user1", "password1", true}, // Valid credentials
            {"user2", "wrongpassword", false}, // Invalid credentials
            {"invalidUser", "password2", false}, // Invalid credentials
            {"user3", "password3", true} // Valid credentials
        };
    }

    // Test method that takes username and password as parameters
    @Test(dataProvider = "loginData")
    public void testLogin(String username, String password, boolean expectedResult) {
        // Simulating a login process
        boolean isLoginSuccessful = validateLogin(username, password);
```

```
// Assert the result
if (expectedResult) {
    Assert.assertTrue(isLoginSuccessful, "Login should be successful for " + username);
    System.out.println("Login successful for user: " + username);
} else {
    Assert.assertFalse(isLoginSuccessful, "Login should fail for " + username);
    System.out.println("Login failed for user: " + username);
}
}

// Simulated login validation method
private boolean validateLogin(String username, String password) {
    // In a real scenario, you would validate against a database or authentication service
    // Here we just simulate successful logins for demonstration purposes
    if ((username.equals("user1") && password.equals("password1")) ||
        (username.equals("user3") && password.equals("password3"))) {
        return true; // Valid credentials
    }
    return false; // Invalid credentials
}
}
```

Output :

```
Nov 01, 2024 11:58:16 AM org.testng.log4testng.Logger info
INFO: [Utils] MethodGroupsHelper.sortMethods() took 0 ms.
Nov 01, 2024 11:58:16 AM org.testng.log4testng.Logger info
INFO: [TestNG] Running:
      C:\Users\bhushan\AppData\Local\Temp\testng-eclipse-872605978\testng-customsuite.xml

Nov 01, 2024 11:58:16 AM org.testng.log4testng.Logger info
INFO: [Utils] DynamicGraphHelper.createDynamicGraph() took 6 ms.
Login successful for user: user1
Login failed for user: user2
Login failed for user: invalidUser
Login successful for user: user3
PASSED: prac11.Problem2.testLogin("invalidUser", "password2", false)
PASSED: prac11.Problem2.testLogin("user1", "password1", true)
PASSED: prac11.Problem2.testLogin("user2", "wrongpassword", false)
PASSED: prac11.Problem2.testLogin("user3", "password3", true)

=====
      Default test
      Tests run: 1, Failures: 0, Skips: 0
=====

=====
Default suite
Total tests run: 4, Passes: 4, Failures: 0, Skips: 0
```


3. Create a test class that performs cross-browser testing for a simple web application. Also configure an XML file to specify which browsers to test against.

Code :

```
package prac11;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.testng.annotations.AfterClass;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

public class Problem3 {

    private WebDriver driver;

    @Parameters("browser")
    @BeforeClass
    public void setUp(String browser) {
        if (browser.equalsIgnoreCase("chrome")) {

            System.setProperty("webdriver.chrome.driver", "D:\\selenium_setup\\chromedriver.exe");
            driver = new ChromeDriver();
        } else if (browser.equalsIgnoreCase("firefox")) {

            System.setProperty("webdriver.gecko.driver", "D:\\selenium_setup\\geckodriver.exe");
            driver = new FirefoxDriver();
        }

        // Add other browsers as needed
    }
}
```

```
        driver.manage().window().maximize();
    }

    @Test
    public void testWebApplication() {
        driver.get("https://demoqa.com");
        // You can add more interactions with the web application here
        System.out.println("Page Title: " + driver.getTitle());
        // Add assertions here as needed
    }

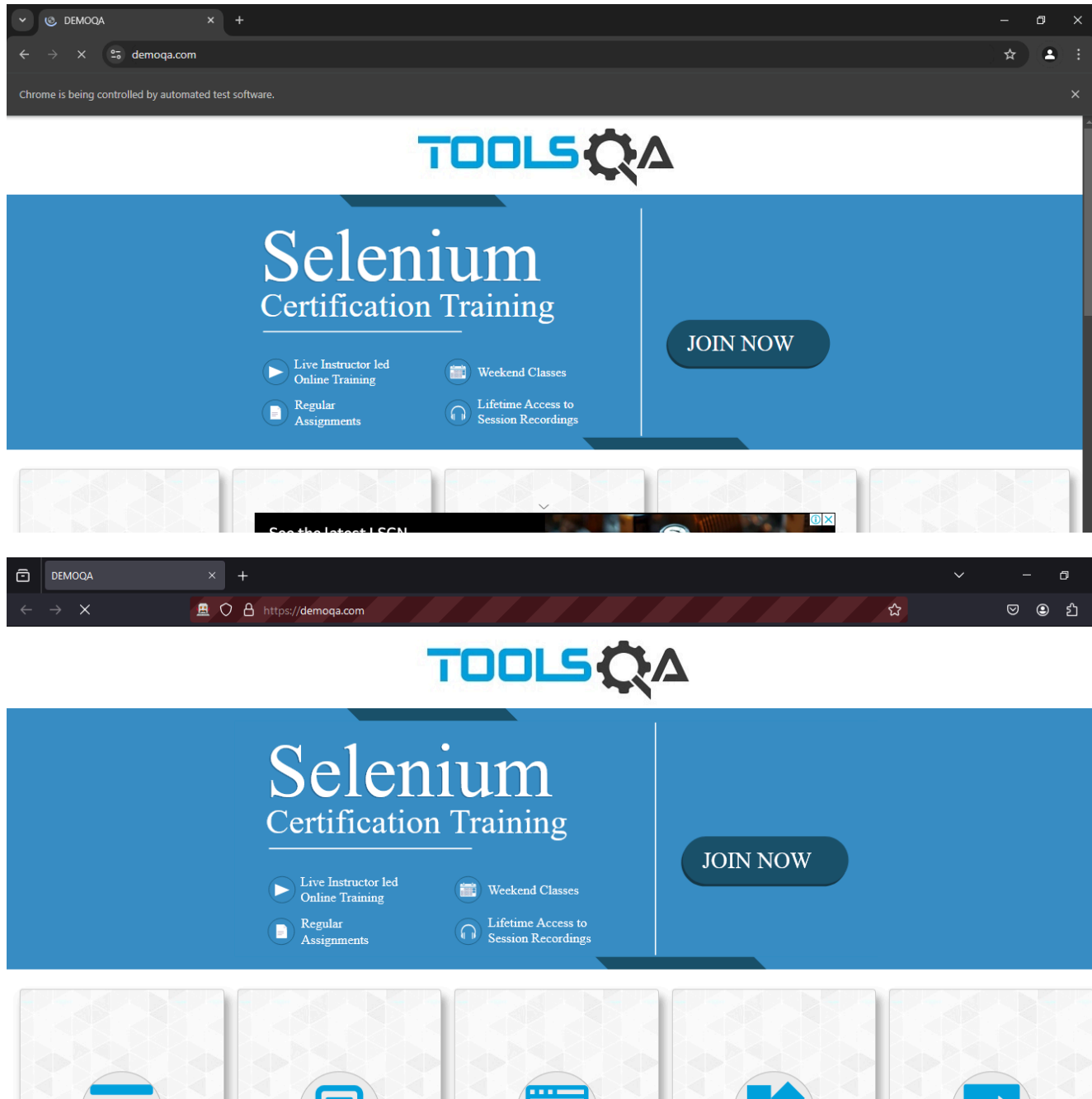
    @AfterClass
    public void tearDown() {
        if (driver != null) {
            driver.quit();
        }
    }
}
```

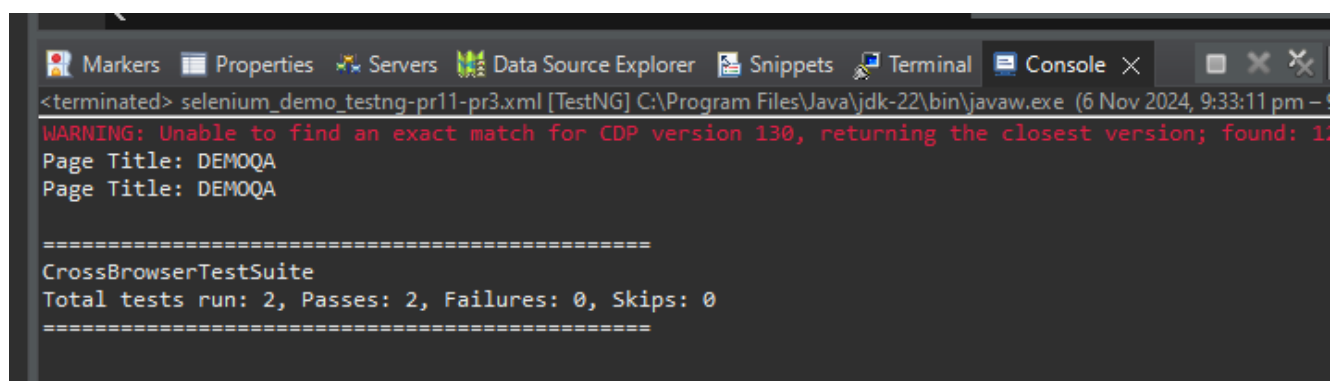
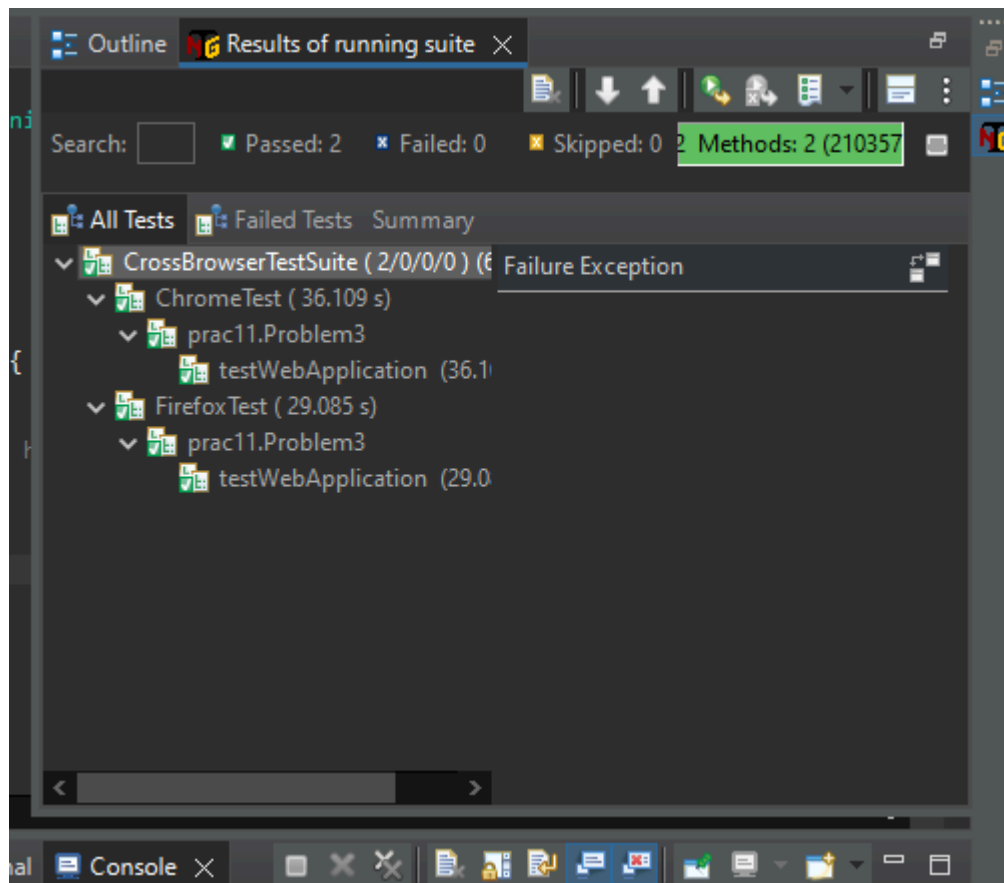
XML File :

```
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite name="CrossBrowserTestSuite">
    <test name="ChromeTest">
        <parameter name="browser" value="chrome"/>
        <classes>
            <class name="prac11.Problem3"/>
        </classes>
    </test>
    <test name="FirefoxTest">
        <parameter name="browser" value="firefox"/>
        <classes>
```

```
<class name="prac11.Problem3"/>
</classes>
</test>
</suite>
```

Output :





Conclusion: Understood how to use Basic Automation Frameworks - linear scripting, library architecture framework, data driven Framework.

After performing this Practical/lab, students are expected to answer following questions

Q.1 What is @DataProvider Annotations?

Q.2 What is Cross Browser Testing?