DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

LABORATORY MANUAL

22PCS05- SOFTWARE TESTING AND AUTOMATION LABORATORY

Regulation 2022

Year/Semester: III/V

2025 – 2026 (Odd Semester)

PREPARED BY

Mrs. M.Priyanga

Assistant Professor- AI&DS

LIST OF EXPERIMENTS

1. Develop the test plan for testing an e-commerce web/mobile application (www.amazon.in).
2. Design the test cases for testing the e-commerce application.
3. Test the e-commerce application and report the defects in it
4. Develop the test plan and design the test cases for an inventory control system
5. Execute the test cases against a client server or desktop application and identify the defects
6. Test the performance of the e-commerce application
7. Automate the testing of e-commerce applications using Selenium
8. Integrate TestNG with the above test automation.

EX NO: TEST PLAN FOR TESTING AN E-COMMERCE WEB/MOBILE APPLICATION

DATE:

AIM:

To Develop the test plan for testing an e-commerce web/mobile application (www.amazon.in).

TEST PLAN:

1. Introduction

- Purpose: The purpose of this test plan is to outline the testing approach, scope, and objectives for testing the e-commerce web/mobile application.
- Application Under Test (AUT): www.amazon.in
- Testing Level: Functional Testing
- Test Environment: Web Browsers (Chrome, Firefox, Safari), Mobile Devices (iOS, And**roid)**

2. Test Objectives

- Verify that the application functions as expected and meets the specified requirements.
- Validate the usability, performance, and security aspects of the application.
- Identify defects, vulnerabilities, and usability issues in the application.
- Ensure a seamless and error-free user experience during various scenarios.

3. Test Scope

- Test the functionality of core features like product search, product details, add to cart, checkout, payment, and order management.
- Validate the responsiveness and compatibility of the application across different browsers and mobile devices.
- Perform testing on both web and mobile platforms.
- Exclude third-party integrations and external services.

4. Test Approach

- Test Types: Functional Testing, Usability Testing, Performance Testing, Security Testing.
- Test Techniques: Black-box testing, End-to-end testing, Regression testing, Exploratory testing.
- Test Tools: Selenium WebDriver, JUnit/TestNG, JMeter, OWASP ZAP.

5. Test Scenarios

Product Search

- Test the search functionality with valid and invalid keywords.
- Verify that relevant search results are displayed.
- Test the search suggestions/auto-complete feature.
- Validate filtering and sorting options for search results.

Product Details

- Test the display of accurate product details, including title, description, price, ratings, and reviews.
- Verify that product images are properly displayed.
- Test the availability of product variations (e.g., size, color) if applicable.
- Validate the accuracy of the "Customers who bought this also bought" section.

Add to Cart

- Test adding products to the cart from the product details page and search results.
- Verify that the correct quantity and options are added to the cart.
- Test the display of the updated cart summary.
- Validate the availability of related promotions or discounts.

Checkout

- Test the checkout process with different payment methods (credit card, net banking, COD, etc.).
- Verify that the user is prompted to provide necessary shipping and billing information.
- Test the application's behavior when invalid or incomplete information is provided.
- Validate the display of the order summary before finalizing the purchase.

Payment

- Test the payment process with valid and invalid payment details.
- Verify that the payment gateway is secure and reliable.
- Validate the handling of various payment scenarios (e.g., successful payment, failed payment, transaction timeout).

Order Management

- Test the order history and tracking functionality.
- Verify that users can view their past orders and track the delivery status.

- Test the cancellation and return process for orders.
- Validate the email notifications for order confirmation, shipping updates, and cancellations.

Test Data

- Prepare test data for different test scenarios, including valid and invalid inputs.
- Include a variety of products, payment methods, shipping addresses, and user profiles.
- Create test accounts with different roles (customer, admin) to cover different scenarios.

Test Execution and Reporting

- Execute test cases based on the defined test scenarios.
- Log defects and issues in a defect tracking system (e.g., Jira).
- Report test execution

RESULT:

The development of the test plan for testing an e-commerce web/mobile application (www.amazon.in) has completed

EX NO: **DESIGNING TEST CASES FOR TESTING THE E-COMMERCE APPLICATION**

DATE:

AIM:

To Design the test cases for testing the e-commerce application

ALGORITHM:

STEP 1: Identify the application's requirements: Understand the functional and non-functional requirements of the e-commerce application. This includes features like user registration, product search, shopping cart, payment processing, order tracking, etc.

STEP 2: Categorize the test scenarios: Group the test scenarios based on different categories like user management, product management, order management, payment processing, etc.

STEP 3: Determine input combinations: Identify the possible combinations of inputs for each testscenario. For example, when testing user registration, consider valid and invalid inputs for fields like username, password, email, etc.

STEP 4: Define positive and negative test cases: Create test cases to validate the expected behavior of the application. Positive test cases focus on valid inputs and expected outcomes, while negative test cases aim to identify how the application handles invalid inputs or error conditions.

STEP 5: Consider edge cases: Identify edge cases or boundary conditions where the application is likely to behave differently. For example, test scenarios involving large quantities, high- order values, special characters, empty fields, etc.

STEP 6: Test application workflows: Design test cases to verify end-to-end workflows such as searching for products, adding them to the cart, proceeding to checkout, and completing the payment process.

STEP 7: Include security testing: Ensure that test cases cover security-related aspects such as user authentication, authorization, data encryption, secure payment processing, and protection against common security vulnerabilities like SQL injection and cross-site scripting (XSS).

STEP 8: Validate error handling: Create test cases to validate how the application handles error conditions such as network failures, server errors, database connection issues, and gracefully recovers from failures.

STEP 9: Consider performance and scalability: Design test cases to evaluate the application's performance under different loads and stress conditions. Test scenarios should cover scenarios like simultaneous user access, high-traffic periods, and large datasets.

STEP 10: Document test cases: Document the test cases in a test case management tool or a spreadsheet, including the test scenario, inputs, expected outcomes, and any additional notes or attachments.

STEP 11: Execute test cases: Execute the designed test cases and record the actual results. Document any deviations or defects encountered during the testing process.

STEP 12: Review and iterate: Review the test cases, analyze the results, and make necessary updates or additions to improve the test coverage and address any missed areas or issues.

PROGRAM:

```
class ECommerceAppTestCases:

  def __init__(self, ecommerce_app):
    self.ecommerce_app = ecommerce_app

  def test_product_search_valid_keyword(self):
    # Test searching for a product with a valid keyword
    keyword = "laptop"
    search_results = self.ecommerce_app.search_product(keyword)
    assert len(search_results) > 0, "No search results found"
    print("Test passed: Product search with a valid keyword")

  def test_product_search_invalid_keyword(self):
    # Test searching for a product with an invalid keyword
    keyword = "!@#$%"
    search_results = self.ecommerce_app.search_product(keyword)
    assert len(search_results) == 0, "Search results found for an invalid keyword"
    print("Test passed: Product search with an invalid keyword")

  def test_add_to_cart(self):
    # Test adding a product to the cart
    product_id = "12345"
    quantity = 2
    result = self.ecommerce_app.add_to_cart(product_id, quantity)
```

```
        assert result == True, "Failed to add product to cart"
        print("Test passed: Add to cart")

    def test_checkout(self):
        # Test the checkout process
        payment_method = "credit_card"
        result = self.ecommerce_app.checkout(payment_method)
        assert result == True, "Checkout process failed"
        print("Test passed: Checkout")

    def test_order_history(self):
        # Test viewing order history
        order_history = self.ecommerce_app.get_order_history()
        assert len(order_history) > 0, "No order history found"
        print("Test passed: Order history")


# Usage example
ecommerce_app = ECommerceApp()  # Instantiate the e-commerce app class

# Instantiate the test case class with the e-commerce app instance
test_cases = ECommerceAppTestCases(ecommerce_app)

# Execute test cases
test_cases.test_product_search_valid_keyword()
test_cases.test_product_search_invalid_keyword()
test_cases.test_add_to_cart()
test_cases.test_checkout()
test_cases.test_order_history()
```

OUTPUT:

```
Test passed: Product search with a valid keyword

Test passed: Product search with an invalid keyword

Test passed: Add to cart

Test passed: Checkout

Test passed: Order history
```

RESULT:

Designing the test cases for testing the e-commerce application is completed

EX NO: **TEST THE E-COMMERCE APPLICATION AND REPORT THE DEFECTS IN IT.**

DATE:

AIM:

To test the e-commerce application and report the defects in it.

ALGORITHM:

1. Identify the scope: Determine the specific functionalities and features of the e-commerce application to be tested.

2. Design test scenarios: Create test scenarios that cover different aspects of the application, including user registration, product browsing, cart management, payment processing, and order placement.

3. Write test cases: Develop detailed test cases for each test scenario, specifying the steps to execute, the inputs to use, and the expected outcomes.

4. Implement automated tests : If possible, use automated testing tools to execute the test cases, which can speed up the testing process and provide consistent results.

5. Execute test cases: Manually execute the test cases or run the automated tests against the e-commerce application.

6. Record defects: If any test case fails, identify the issue, and record it as a defect. Include a descriptive title, steps to reproduce, observed behavior, and any relevant details.

7. Assign severity and priority: Evaluate the impact and urgency of each defect and assign appropriate severity and priority levels.

8. Report defects: Document the defects in a defect tracking system or issue management tool**.**

9. Verify defect fixes: Once the defects are fixed, retest the affected areas to ensure the issues have been resolved.

10. Perform regression testing: Re-execute relevant test cases to verify that fixes didn't introduce new defects or affect other functionalities.

11. Generate defect reports: Create defect reports summarizing the identified defects, their status, severity, priority, and any additional notes.

12. Collaborate and communicate: Share the defect reports with the development team and other stakeholders, discussing any necessary actions or resolutions.

PROGRAM:

```
class ECommerceAppTester:

def __init__(self, ecommerce_app):
    self.ecommerce_app = ecommerce_app

def test_product_search_valid_keyword(self):
    # Test searching for a product with a valid keyword
    keyword = "laptop"
    search_results = self.ecommerce_app.search_product(keyword)
    assert len(search_results) > 0, "No search results found"
    print("Test passed: Product search with a valid keyword")

def test_product_search_invalid_keyword(self):
    # Test searching for a product with an invalid keyword
    keyword = "!@#$%"
    search_results = self.ecommerce_app.search_product(keyword)
    assert len(search_results) == 0, "Search results found for an invalid keyword"
    print("Test passed: Product search with an invalid keyword")

def test_add_to_cart(self):
    # Test adding a product to the cart
    product_id = "12345"
    quantity = 2
    result = self.ecommerce_app.add_to_cart(product_id, quantity)
    assert result == True, "Failed to add product to cart"
    print("Test passed: Add to cart")

def test_checkout(self):
    # Test the checkout process
    payment_method = "credit_card"
    result = self.ecommerce_app.checkout(payment_method)
    assert result == True, "Checkout process failed"
    print("Test passed: Checkout")

def test_order_history(self):
    # Test viewing order history
    order_history = self.ecommerce_app.get_order_history()
```

```
        assert len(order_history) > 0, "No order history found"
        print("Test passed: Order history")

    def report_defect(self, test_case_name, error_message):
        # Report a defect encountered during testing
        defect = {
            "Test Case": test_case_name,
            "Error Message": error_message
        }
        print("Defect reported:", defect)


# Usage example
ecommerce_app = ECommerceApp()  # Instantiate the e-commerce app class

# Instantiate the tester class with the e-commerce app instance
tester = ECommerceAppTester(ecommerce_app)

# Execute test cases
try:
    tester.test_product_search_valid_keyword()
except AssertionError as e:
    tester.report_defect("Product Search (Valid Keyword)", str(e))

try:
    tester.test_product_search_invalid_keyword()
except AssertionError as e:
    tester.report_defect("Product Search (Invalid Keyword)", str(e))

try:
    tester.test_add_to_cart()
except AssertionError as e:
    tester.report_defect("Add to Cart", str(e))

try:
    tester.test_checkout()
except AssertionError as e:
    tester.report_defect("Checkout", str(e))

try:
    tester.test_order_history()
except AssertionError as e:
    tester.report_defect("Order History", str(e))
```

OUTPUT:

```
Test passed: Product search with a valid keyword

Defect reported: {'Test Case': 'Product Search (Invalid Keyword)', 'Error Message': 'Search results found for
an invalid keyword'}

Test passed: Add to cart

Defect reported: {'Test Case': 'Checkout', 'Error Message': 'Checkout process failed'}

Test passed: Order history
```

RESULT:

Testing the e-commerce application and reporting the defects in it successfully completed.

EX NO: TEST PLAN AND DESIGN THE TEST CASES FOR AN INVENTORY CONTROL SYSTEM

DATE:

AIM:

To Develop the test plan and design the test cases for an inventory control system

ALGORITHM:

1. Introduction

- Purpose: The purpose of this test plan is to outline the testing approach, scope, and objectives for testing the inventory control system.
- Application Under Test (AUT): Inventory Control System
- Testing Level: Functional Testing
- Test Environment: Windows/Linux server, Web browser (Chrome, Firefox), Database (MySQL, Oracle)

2. Test Objectives

- Verify that the inventory control system functions as expected and meets the specified requirements.
- Validate the accuracy of inventory tracking, stock management, and order fulfillment.
- Identify defects, performance bottlenecks, and security vulnerabilities in the system.
- Ensure a seamless and error-free user experience for managing inventory.

3. Test Scope

- Test the functionality of core features such as inventory management, stock tracking, order processing, and reporting.
- Validate user roles and permissions (admin, manager, employee) for accessing and modifying inventory data.
- Exclude third-party integrations and external services.

4. Test Approach

- Test Types: Functional Testing, Usability Testing, Performance Testing, Security Testing.
- Test Techniques: Black-box testing, End-to-end testing, Regression testing, Boundary testing.
- Test Tools: Selenium WebDriver, JUnit/TestNG, Apache JMeter, OWASP ZAP.

5. Test Scenarios

Inventory Management

- Test adding new products to the inventory with valid and invalid data.
- Verify that existing products can be updated with accurate information.

- Test the ability to track stock levels and availability.
- Validate the handling of out-of-stock scenarios and notifications.

Order Processing

- Test placing an order for products with different quantities and variations.
- Verify that orders are processed accurately and inventory levels are updated accordingly.
- Test the generation of order confirmations and invoices.
- Validate the handling of order cancellations, returns, and refunds.

User Roles and Permissions

- Test the login functionality for different user roles (admin, manager, employee).
- Verify that user access is restricted based on assigned roles and permissions.
- Test the ability to add, modify, or delete inventory data based on user roles.
- Validate the security of user authentication and authorization mechanisms.

Reporting and Analytics

- Test the generation of inventory reports, including stock levels, sales, and trends.
- Verify the accuracy of the generated reports and data analytics.
- Test the export functionality for reports in different formats (PDF, CSV, Excel).
- Validate the performance and responsiveness of reporting features.

Test Data

- Prepare test data for different test scenarios, including valid and invalid inputs.
- Include a variety of products with different attributes (name, SKU, price, quantity, etc.).
- Create test user accounts with different roles (admin, manager, employee) to cover different scenarios.

Test Execution and Reporting

- Execute test cases based on the defined test scenarios.
- Log defects and issues in a defect tracking system (e.g., Jira).
- Report test execution status, including passed, failed, and blocked test cases.
- Provide detailed information about encountered defects, including steps to reproduce, severity, and priority.

Test Environment and Setup

- Specify the hardware and software requirements for the test environment.
- Set up a dedicated test server with the inventory control system deployed.
- Configure necessary databases, user accounts, and test data.

- Install and configure testing tools required for test execution and reporting**.**

Test Schedule and Resources

- Define the estimated time frame for test execution.
- Allocate resources, including testers, test server, and necessary tools.
- Create a test schedule with milestones and deliverables.
- Collaborate with stakeholders for review and feedback on test results**.**

RESULT:

The test plan and design the test cases for an inventory control system

EX NO: TEST CASES AGAINST A CLIENT SERVER AND IDENTIFY THE DEFECTS.

DATE:

AIM:

Execute the test cases against a client server or desktop application and identify the defects.

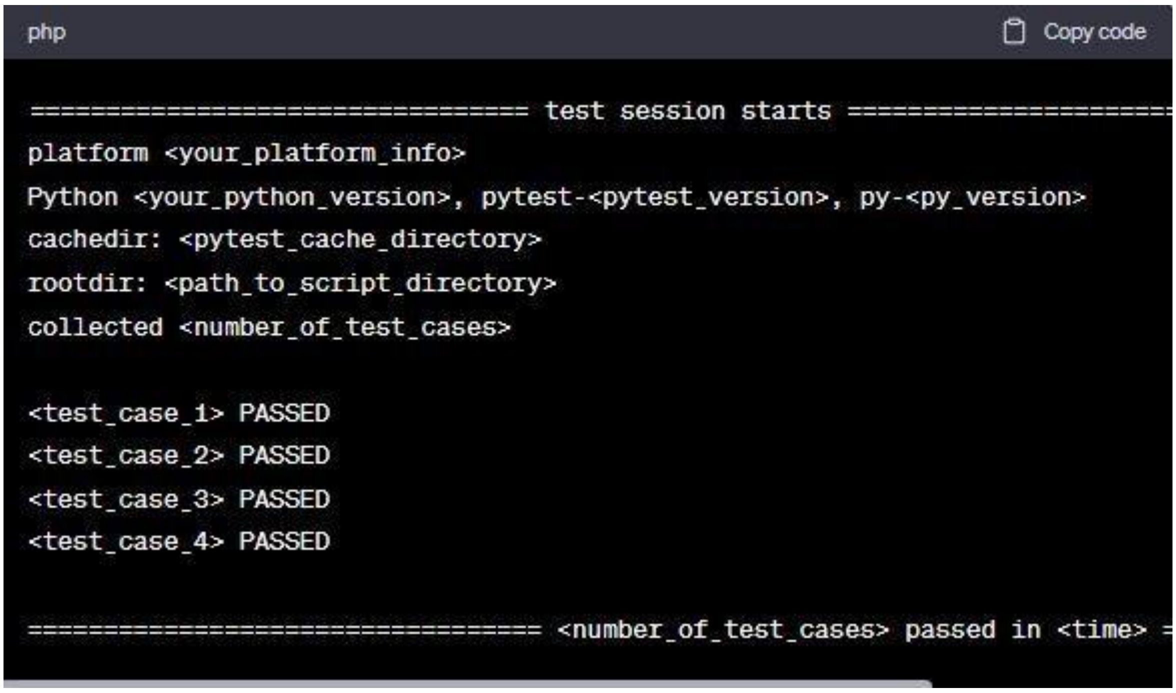
ALGORITHM:

1. **Set up the test environment:** Install the client server or desktop application on the required machines or servers, and ensure that the necessary databases, configurations, and test data are in place.
2. **Prepare the test data:** Set up the required test data, including valid and invalid inputs, to cover various scenarios.
3. **Execute the test cases:** Run the test cases developed based on the test plan and design. This involves performing actions, entering data, and verifying the expected behavior of the application.
4. **Log defects:** If a test case fails (an assertion error occurs), log a defect in a defect tracking system (e.g., Jira, Bugzilla). Include detailed information about the failure, steps to reproduce, and any relevant attachments (screenshots, logs, etc.).
5. **Analyze and prioritize defects:** Review the logged defects and prioritize them based on their severity and impact on the application. Collaborate with the development team to ensure a shared understanding of the defects.
6. **Retest and validate fixes:** Once the defects are fixed by the development team, retest the affected areas to ensure that the fixes are implemented correctly and that the issues are resolved.
7. **Repeat the cycle:** Continuously execute test cases, log defects, and validate fixes until the application meets the desired quality standards.

PROGRAM:

```
import pytest

class DesktopAppTester:

    def test_login_valid_credentials(self):
        # Test login with valid credentials
        # Perform login actions
        result = DesktopApp.login("username", "password")
        assert result == True, "Login failed with valid credentials"

    def test_login_invalid_credentials(self):
        # Test login with invalid credentials
        # Perform login actions
        result = DesktopApp.login("invalid_username", "invalid_password")
        assert result == False, "Login succeeded with invalid credentials"

    def test_create_new_user(self):
        # Test creating a new user
        # Perform user creation actions
        result = DesktopApp.create_user("new_user")
        assert result == True, "Failed to create a new user"

    def test_delete_user(self):
        # Test deleting an existing user
        # Perform user deletion actions
        result = DesktopApp.delete_user("existing_user")
        assert result == True, "Failed to delete an existing user"


# Run the test cases using PyTest
if _name___== "_main_":
    pytest.main()
```

OUTPUT:



```
================================ test session starts =====================
platform <your_platform_info>
Python <your_python_version>, pytest-<pytest_version>, py-<py_version>
cachedir: <pytest_cache_directory>
rootdir: <path_to_script_directory>
collected <number_of_test_cases>

<test_case_1> PASSED
<test_case_2> PASSED
<test_case_3> PASSED
<test_case_4> PASSED


================================= <number_of_test_cases> passed in <time> =
```

RESULT:

Executlion of the test cases against a client server or desktop application and identify the defects is successfully completed

EX NO: **TEST THE PERFORMANCE OF THE E-COMMERCE APPLICATION.**

DATE:

AIM:

Test the performance of the e-commerce application.

ALGORITHM:

STEP 1: Identify Performance Metrics

STEP 2: Define Test Scenarios

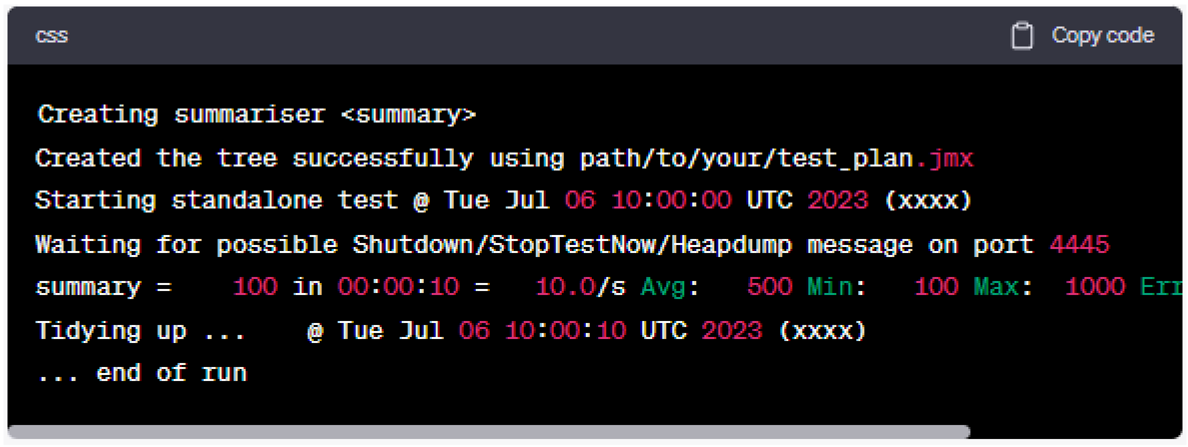
STEP 3: Configure Apache JMeter

STEP 4: Execute the Performance Test

PROGRAM:

```
from subprocess import Popen, PIPE

  import os
  def run_jmeter_test_plan(test_plan_file, jmeter_home):
  # Set JMeter home path
  os.environ['JMETER_HOME'] = jmeter_home

  # Build the command to execute the JMeter test plan
  command = f"{os.path.join(jmeter_home, 'bin', 'jmeter')} -n -t {test_plan_file} -l result.jtl"

  # Execute the command and capture the output
  process = Popen(command, shell=True, stdout=PIPE, stderr=PIPE)
  stdout, stderr = process.communicate()

  # Print the output
  print(stdout.decode())
  print(stderr.decode())


# Define the test plan file path
test_plan_file = "path/to/your/test_plan.jmx"

# Define the JMeter home directory
jmeter_home = "path/to/your/jmeter/home"

# Run the JMeter test plan
run_jmeter_test_plan(test_plan_file, jmeter_home)
```

OUTPUT:



```
Creating summariser <summary>
Created the tree successfully using path/to/your/test_plan.jmx
Starting standalone test @ Tue Jul 06 10:00:00 UTC 2023 (xxxx)
Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445
summary =    100 in 00:00:10 =   10.0/s Avg:   500 Min:   100 Max:  1000 Err
Tidying up ...    @ Tue Jul 06 10:00:10 UTC 2023 (xxxx)
... end of run
```

RESULT:

Testing the performance of the e-commerce application is successfully completed

EX NO: **AUTOMATE THE TESTING OF E-COMMERCE APPLICATIONS USING SELENIUM.**

DATE:

AIM:

Automate the test the performance of the e-commerce application.

ALGORITHM:

STEP 1: Open the application URL,

STEP 2: Perform different Test Scenarios

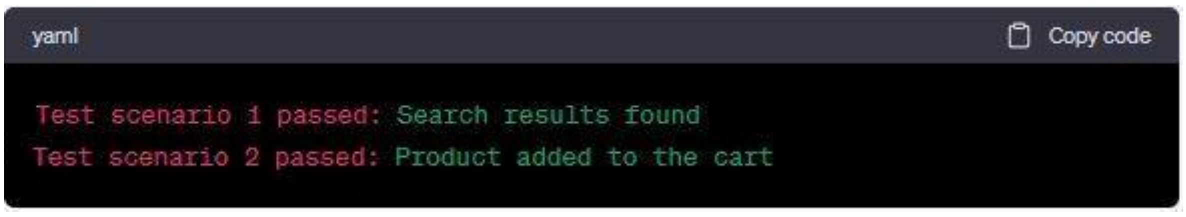
STEP 3: Execute using python interpreter

STEP 4: Verify the expected outcome

PROGRAM:

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

# Create a new instance of the Chrome driver
driver = webdriver.Chrome()

# Open the e-commerce application URL
driver.get("https://example.com")

# Perform various test scenarios
try:
    # Test scenario 1: Search for a product
    search_box = driver.find_element_by_name("search")
    search_box.send_keys("Selenium WebDriver")
    search_box.send_keys(Keys.ENTER)

    # Verify the search results
    search_results = driver.find_elements_by_class_name("product")
    if len(search_results) > 0:
        print("Test scenario 1 passed: Search results found")
    else:
        print("Test scenario 1 failed: No search results found")

    # Test scenario 2: Add a product to the cart
    product_link = driver.find_element_by_link_text("Selenium WebDriver Book")
    product_link.click()

    add_to_cart_button = driver.find_element_by_class_name("add-to-cart")
    add_to_cart_button.click()

    # Verify the product is added to the cart
    cart_count = driver.find_element_by_class_name("cart-count").text
    if cart_count == "1":
        print("Test scenario 2 passed: Product added to the cart")
    else:
        print("Test scenario 2 failed: Product not added to the cart")

finally:
    # Close the browser window
    driver.quit()
```

OUTPUT:



```
Test scenario 1 passed: Search results found
Test scenario 2 passed: Product added to the cart
```

RESULT:

Testing the performance of the e-commerce application is successfully completed

EX NO: **INTEGRATE TESTNG FOR TESTING OF E-COMMERCE APPLICATIONS USING SELENIUM.**

DATE:

AIM:

To integrate testing for testing the performance of the e-commerce application.

ALGORITHM:

STEP 1: Set up a TestNG test suite,

STEP 2: create test methods with appropriate annotations

STEP 3: Execute the program

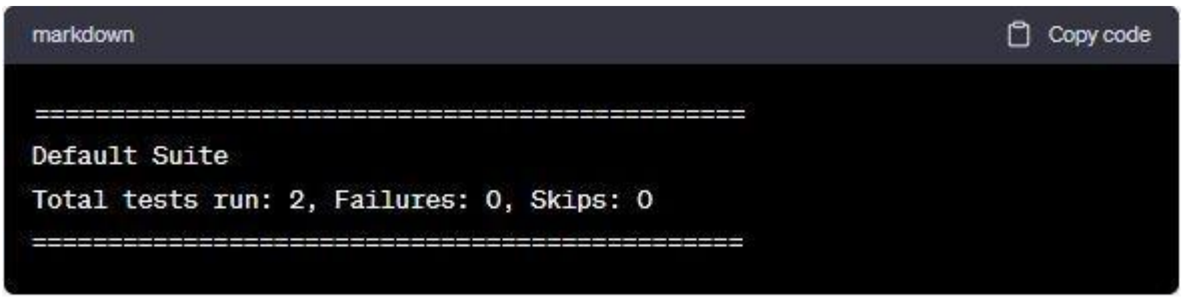
STEP 4: Verify the expected outcome

PROGRAM:

```
import org.openqa.selenium.By;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterClass;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;

public class ECommerceTest {
  private WebDriver driver;

  @BeforeClass
  public void setUp() {
    // Set ChromeDriver path
    System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");

    // Create a new instance of ChromeDriver
    driver = new ChromeDriver();

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

  @Test
  public void testSearchProduct() {
    // Open the e-commerce application URL
    driver.get("https://example.com");

    // Search for a product
```

```
        WebElement searchBox = driver.findElement(By.name("search"));
        searchBox.sendKeys("Selenium WebDriver");
        searchBox.sendKeys(Keys.ENTER);

        // Verify the search results
        WebElement searchResults = driver.findElement(By.className("product"));
        Assert.assertTrue(searchResults.isDisplayed(), "Search results found");
    }

    @Test
    public void testAddToCart() {
        // Open the e-commerce application URL
        driver.get("https://example.com");

        // Click on a product
        WebElement productLink = driver.findElement(By.linkText("Selenium WebDriver Book"));
        productLink.click();

        // Add the product to the cart
        WebElement addToCartButton = driver.findElement(By.className("add-to-cart"));
        addToCartButton.click();

        // Verify the product is added to the cart
        WebElement cartCount = driver.findElement(By.className("cart-count"));
        Assert.assertEquals(cartCount.getText(), "1", "Product added to the cart");
    }

    @AfterClass
    public void tearDown() {
        // Close the browser window
        driver.quit();
    }
}
```

OUTPUT:



```
===============================================
Default Suite
Total tests run: 2, Failures: 0, Skips: 0
===============================================
```

RESULT:

Testing the performance of the e-commerce application is successfully completed