Contents

1. Introduction…….. 2
2. Test Automation Framework…….. 2
3. WebShop Automation Framework Design……4
    a. WebShop automation (Cucumber Test Driver & UI Driver) ……...5
    b. Selenium Wrapper……...8
    c. Spring Context……8
4. Pre-Requisites ……...9
5. Test Suite Execution ……...9
    a. Command line……...10
    b. From Eclipse……...10
    c. From Jenkins……...13
6. Cucumber Reports……15
7. Page Object Pattern ……16
8. Object Repository……..18
9. Writing Feature Files ……19
10. Developing Glue code / Step definition……..22
11. Creating a Sample Test ……..……23
12. Appendix: Resources ……..……31

1. Introduction :

Executing regression test scenarios manually can be laborious and time consuming. There will also be some common human errors in cases such as entering test data during monotonous manual testing. Test automation helps in overcoming all these issues. Automated tests developed once can be quickly executed on the application under test deployed on different environments.

Automated tests are developed using available tools and libraries. Tool used may vary depending on the application under test. There are commercially available as well as open source tools. Selenium WebDriver is an open source tool available for web application testing.

For automation of WebShop regression tests, Selenium WebDriver is used along with Cucumber and Java. Test automation framework developed handles execution of test suites on different environments and generates user friendly reports as well as logs.

Some of the advantages of automated software testing are:

a) Improved accuracy
b) Increased test coverage
c) Saves time and money

2. Test Automation Framework:

The automation framework provides the basis of test automation and simplifies the automation effort. The main advantage of a framework is that provides assumptions, concepts and tools that provide support for automated software testing is the low cost for maintenance.

A test automation framework is an execution environment for automated tests. It is the overall system in which the tests will be automated.

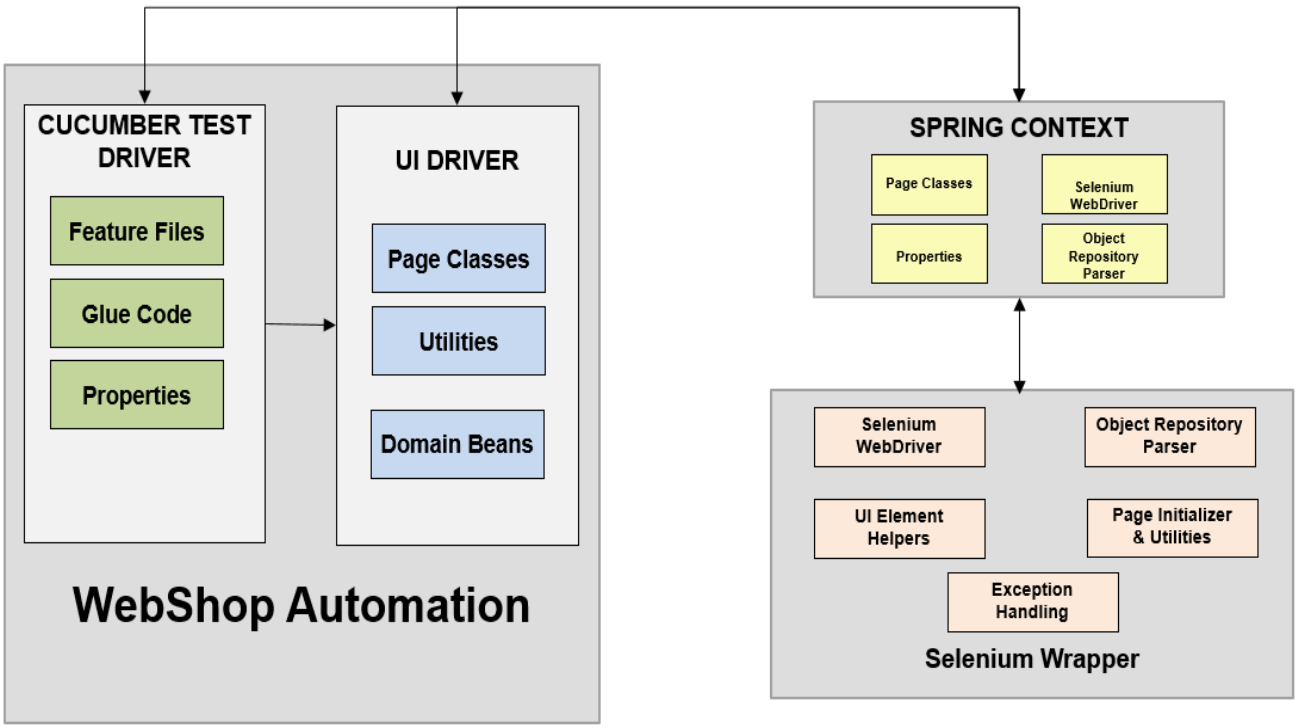
The Testing framework is responsible for:

a) Creating a mechanism to drive test execution
b) Handle data and scripts separately
c) High extensibility
d) Low maintenance
e) Maximize reusability
f) Follow coding standards
g) User friendly reports and logs

Properties of a testing framework:

a) It is application independent.
b) It is easy to expand and maintain.

3. WebShop Automation Framework Design:



WebShop Automation Framework modules are explained below:

WebShop Automation (Cucumber Test Driver & UI Driver):

Cucumber Test Driver:

This includes feature files, glue code and properties files.

Feature Files:

A set of scenarios are written in different feature files. For example all shopping cart scenarios can be included in “ShoppingCart.feature”. The same way different feature files are created for various features.

Example:

```
@webshop @shoppingCart
Feature: Shopping Cart Functionality

##########################################################################################################
@numericQuantityShoppingCart @deleteActiveShoppingCart
Scenario Outline: Check whether changing quantities at shopping cart is set correctly
Given a user with following details is connected to the application:
|organizationType   |webShop    |
|DirectCustomer     |<webShop>  |
Given the user creates a "shared" shopping cart with "cartForTesting"
And the user adds the following product items to cart and navigates to cart:
|itemNo |unit   |quantity   |
|1      |EA     |10         |
|2      |MTR    |1.25       |
When the user changes quantity to "<quantity>" for item "1" in shopping cart
Then the Checkout/Save button should change to "<saveButton>" button
When the user clicks on Save button
Then the Checkout/Save button should change to "<checkOutButton>" button
And the quantity should be set to the entered value "<quantity>" for item "1" in shopping cart

Examples:
|webShop        |checkOutButton         |saveButton     |quantity   |
|KrampUK        |Checkout               |Save           |500        |
```

Every “.feature” file conventionally consists of a single feature. A line starting with the keyword “Feature” followed by free indented text starts a feature.

Every scenario consists of a list of steps, which must start with one of the keywords “Given, When, Then, But, or And”.

Glue Code:

Every step written in the feature file needs to be matching with the respective Java code written in a step definition.

Example:

```
@When("^the user changes quantity to \"(.*?)\" for item \"(.*?)\" in shopping cart$")
public void editItemQuantityInShoppingCart(String quantity, String itemNumber)
        throws AutomationElementNotFoundException {
    originalQuantityValue = shoppingCartPage
            .getQuantiyInCart(sc.getProductItems().get(Integer.parseInt(itemNumber) - 1).getPartNumber());
    shoppingCartPage.enterQuantity(quantity,
            sc.getProductItems().get(Integer.parseInt(itemNumber) - 1).getPartNumber());
}

@When("^the user edits quantity for item in shopping cart to a value to earn free shipping$")
public void editItemQuantityInShoppingCartForFreeShipping() throws NumberFormatException, AutomationElementNotFoundException{
    shoppingCartPage.enterQuantity(calculateAndGetQuantityForFreeShipping(),
            sc.getProductItem().getPartNumber());
}
```

In the above example, there is step definition written for a particular line of a scenario.

Properties Files:

These contain the configurations. Example of a properties file is below:

```
#Selenium WebDriver Properties
################################################################################################
selenium.remoteUrl=http://vsv1e090:4444/wd/hub
selenium.browser=FIREFOX
selenium.operatingSystem=

#Preferences Keys
################################################################################################
firefox.preferences=security.tls.insecure_fallback_hosts,security.tls.version.max,security.tls.version.min,

#Preferences Values - Firefox
################################################################################################
security.tls.insecure_fallback_hosts=string::wcsint.kramp.com
security.tls.version.max=integer::0
security.tls.version.min=integer::0
browser.helperApps.neverAsk.saveToDisk=string::application/xml,text/xml,text/csv,application/octet-stream
browser.download.folderList=integer::2
```

Here the configurations are used for browser setup. Configurations can be environment related, application related or something else which is required for the test execution.

UI Driver:

UI Driver contains the following components:

a) Page classes
b) Utilities
c) Domain beans

Page Classes written for each pages of the application under test. This contains objects on the page as member variables and user's actions as the methods.

Examples: MyAccountPage, ProductContentPage, StoreConfigPage, etc

Utilities contain the utility methods that are used across the project.

a) getCurrentDateWithGivenFormat(String format)
b) clearDirectory(String directory)
c) getLatestFilefromDir(String dirPath)
d) parseStringToDoubleValueWithGivenLocale(String doubleValue, String language)

Domain Beans are the ones' used to hold the data required for the test execution. Some examples of domain beans are User, Order, ProductItem, Store, etc. These are defined under "UIDriver" project.

Selenium Wrapper:

To simplify the development of automated tests, a wrapper is built around the Selenium tool. There are many advantages, some of them are as below:

a. Improves readability of the code
b. Provides standard practices for automated tests development
c. Configurations through properties file
d. Object repository to store object identifiers
e. Meaningful methods to develop the scripts. For example “object.click ()”, “driver.navigateToURL(strURL)”, etc

Below are the modules in Selenium Wrapper:

a. Selenium WebDriver
b. UI Element Helpers
c. Object Repository Parser
d. Page Initializer & Utilities
e. Exception Handling

Spring Context:

The Spring Framework is a very comprehensive framework. The fundamental functionality provided by the Spring Container is dependency injection. Spring provides a light-weight container. This container lets us to inject required objects into other objects. This results in a design in which the Java class are not hard-coupled.

The Spring Container:

- Handles the configuration (e.g. Environment interface for reading all properties files)
- Creates and holds objects of all page classes.

4. Pre-Requisites:

Basic requirements to use this framework are:

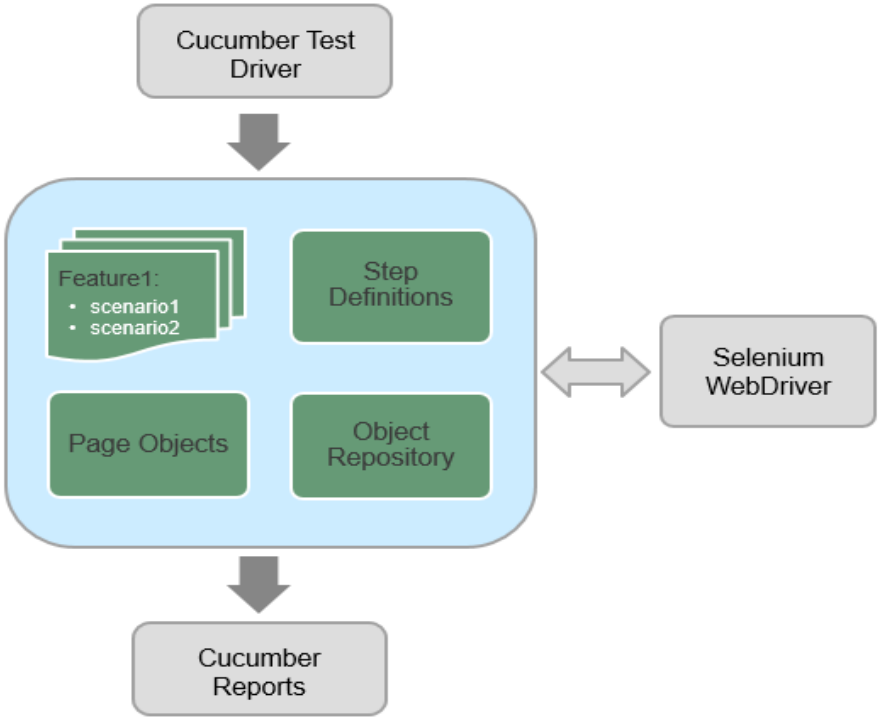
- Eclipse IDE for Java
- Java 1.8

Access to Jenkins (http://vsv1e090:8080/ until a dedicated server is provided for Jenkins) for execution

5. Test Suite Execution

Test scenarios are organized together into features. Each test is given a "tag". Also, tags are provided at Feature level. The tagging is used to group features and scenarios together independent of feature files and directory structure.

Below figure depicts the test execution flow:



Execution of tests start from the “CucumberTest”. This is the test driver which takes the various input parameters such as:

a. Tags (Search.feature, @vatCost, @zeroQuantityShoppingCart)
b. Features (catalogBrowsing.feature, Pricing.feature, Search.feature, etc)
c. Glue (package containing the glue code)
d. Reports directory path

Following are the different ways to start the test execution:

Command line:

Note: Maven should be installed in the system where the command is executed.

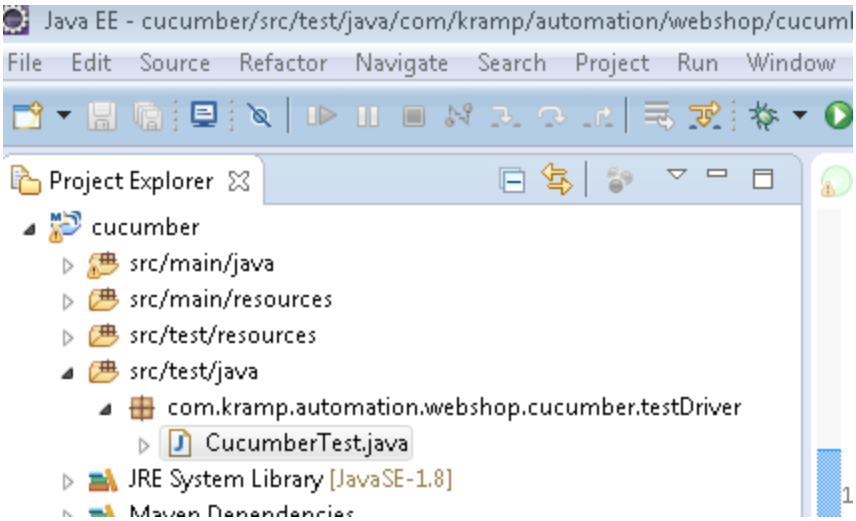
Go to the directory of the WebshopAutomation (\WebShopAutomation\trunk) and then execute the below command:

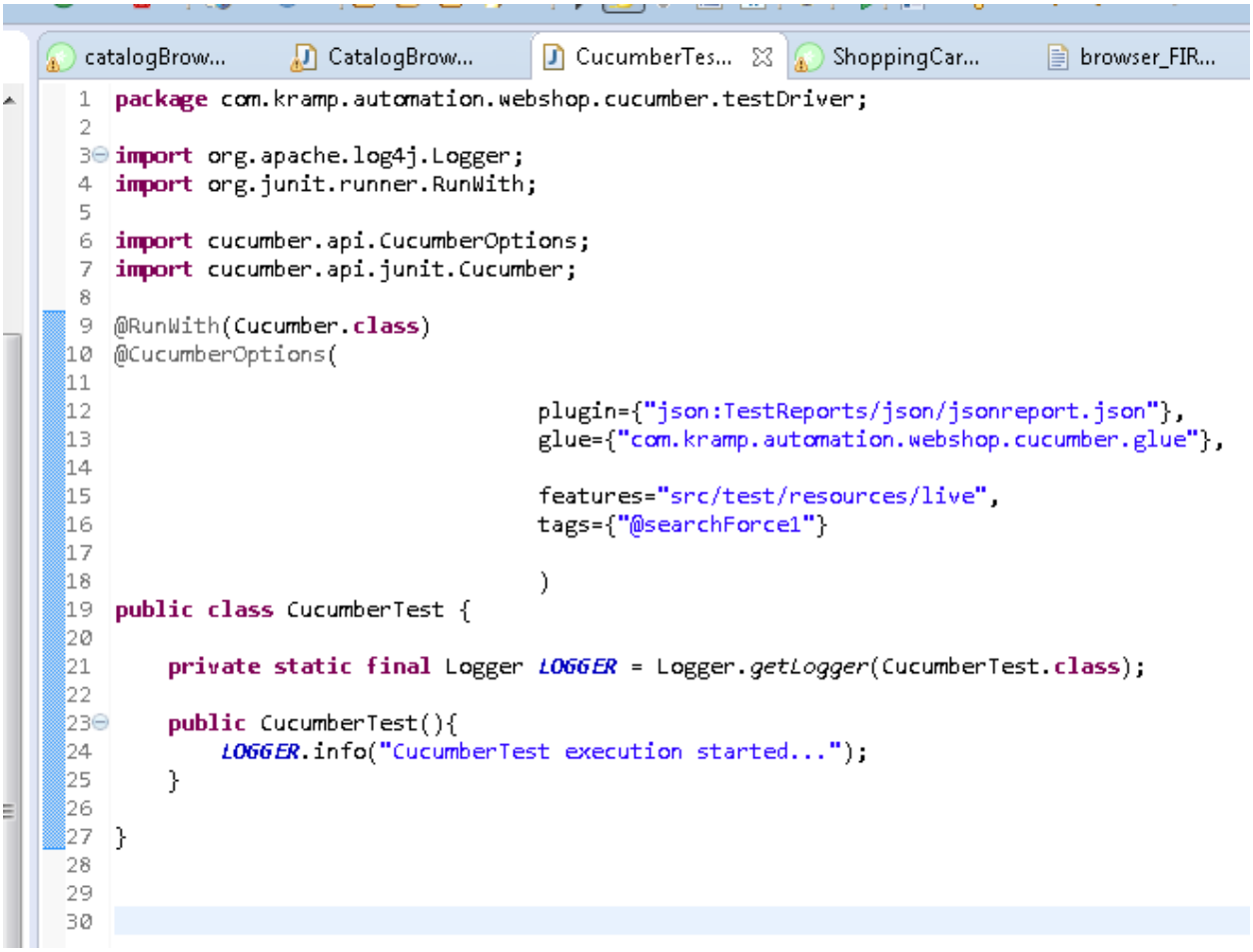
mvn test -DTestEnvironment=qa -DTestBrowser=FIREFOX -Dcucumber.options="--tags @searchForce1"

From Eclipse:

Go to “com.kramp.automation.webshop.cucumber.testDriver” package

Then open “CucumberTest.java” file


Java EE - cucumber/src/test/java/com/kramp/automation/webshop/cucuml
File Edit Source Refactor Navigate Search Project Run Window
Project Explorer
cucumber
src/main/java
src/main/resources
src/test/resources
src/test/java
com.kramp.automation.webshop.cucumber.testDriver
CucumberTest.java
JRE System Library [JavaSE-1.8]


catalogBrow...
CatalogBrow...
CucumberTes...
ShoppingCar...
browser_FIR...
package com.kramp.automation.webshop.cucumber.testDriver;
import org.apache.log4j.Logger;
import org.junit.runner.RunWith;
import cucumber.api.CucumberOptions;
import cucumber.api.junit.Cucumber;
@RunWith(Cucumber.class)
@CucumberOptions(
plugin={"json:TestReports/json/jsonreport.json"},
glue={"com.kramp.automation.webshop.cucumber.glue"},
features="src/test/resources/live",
tags={"@searchForce1"}
)
public class CucumberTest {
private static final Logger LOGGER = Logger.getLogger(CucumberTest.class);
public CucumberTest(){
LOGGER.info("CucumberTest execution started...");
}
}

Setup the TestEnvironment and TestBrowser arguments through “Run Configurations”. Also, provide the cucumber tags to be executed in the tags field.

Below figures (figure -1 & 2) show how to provide the cucumber tags and

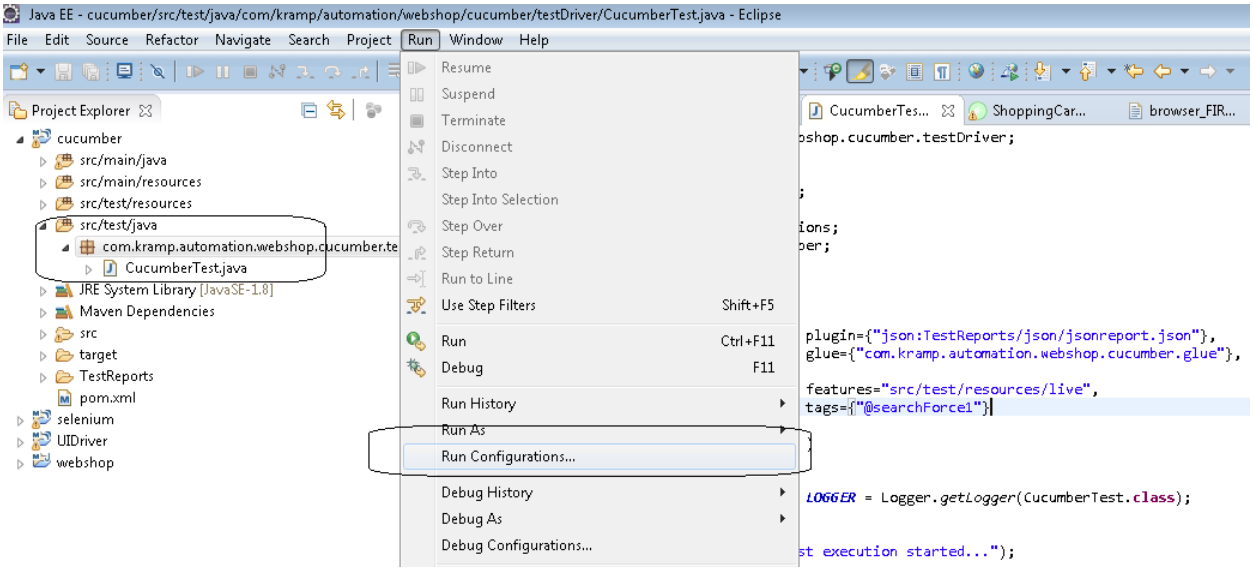


Figure – 1

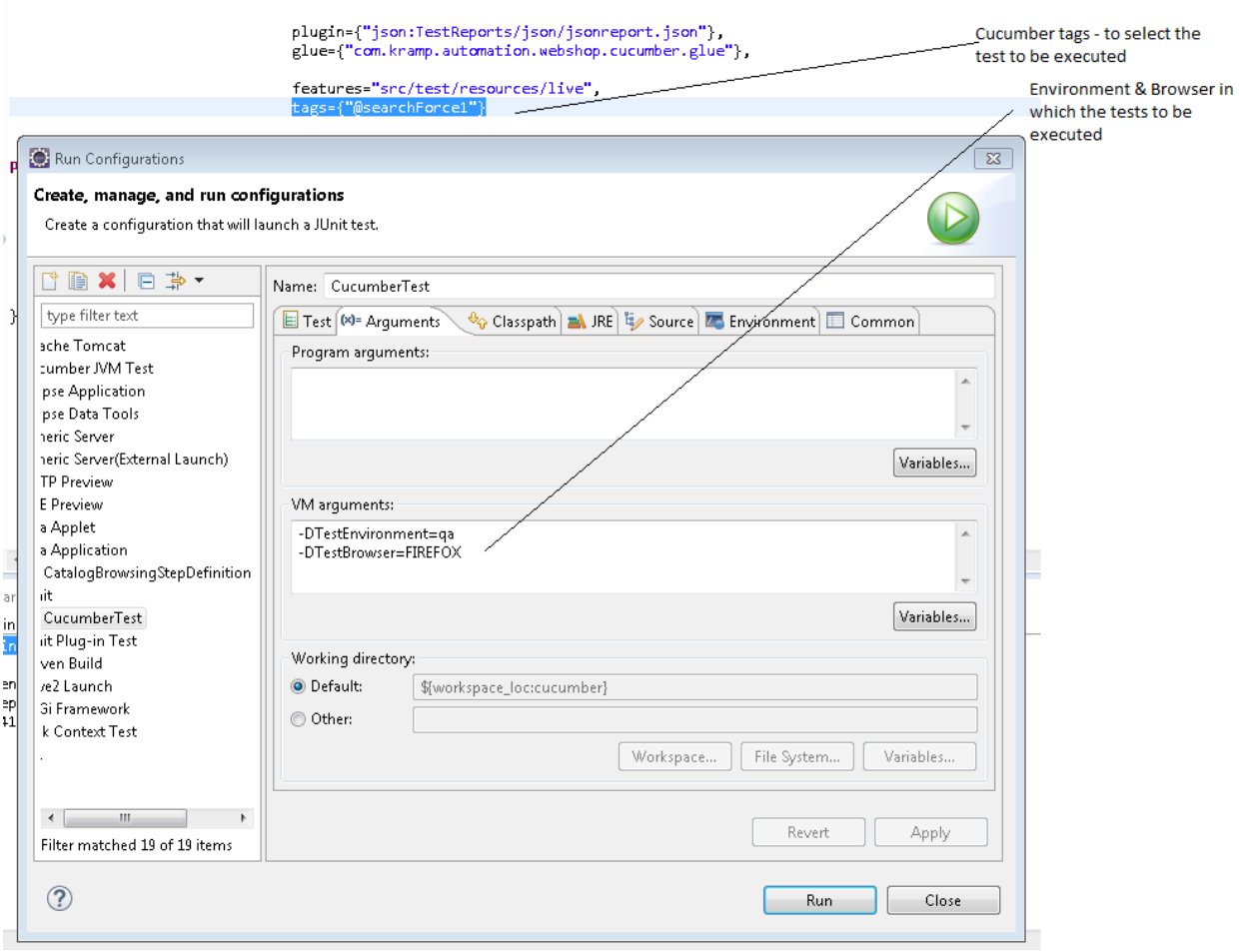


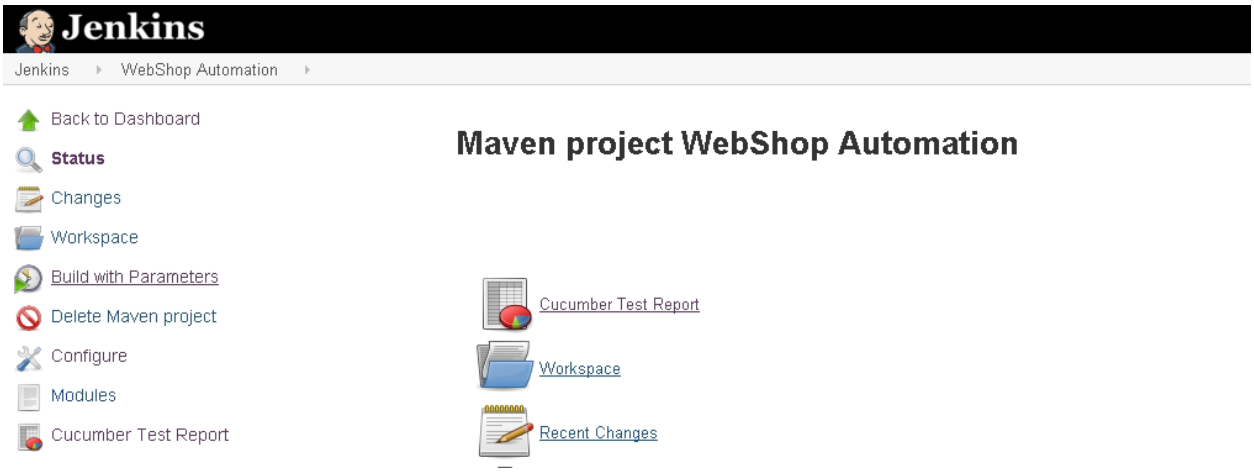
Figure – 2

From Jenkins:

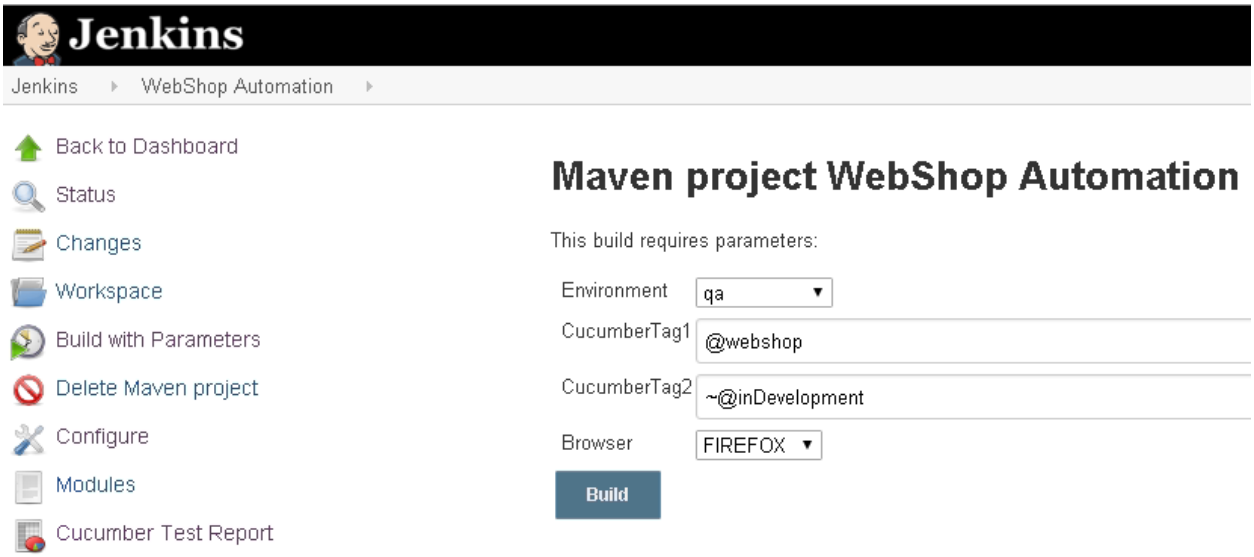
Go to Jenkins using the below URL:

http://{hostname:port}/job/WebShop%20Automation/

(e.g. Current Jenkins build is available at: http://vsv1e090:8080/job/WebShop%20Automation/)



Click “Build with Parameters” and provide the parameters as in the below figure.



Click “Build” to start the test execution. This will pick up all the feature files which are tagged with “webshop” and then executes all the scenarios.

Once the test execution is done, click “Cucumber Test Report” and report gets opened.

6. Cucumber Reports:

As shown in the above figures Cucumber Report contains the information such as:

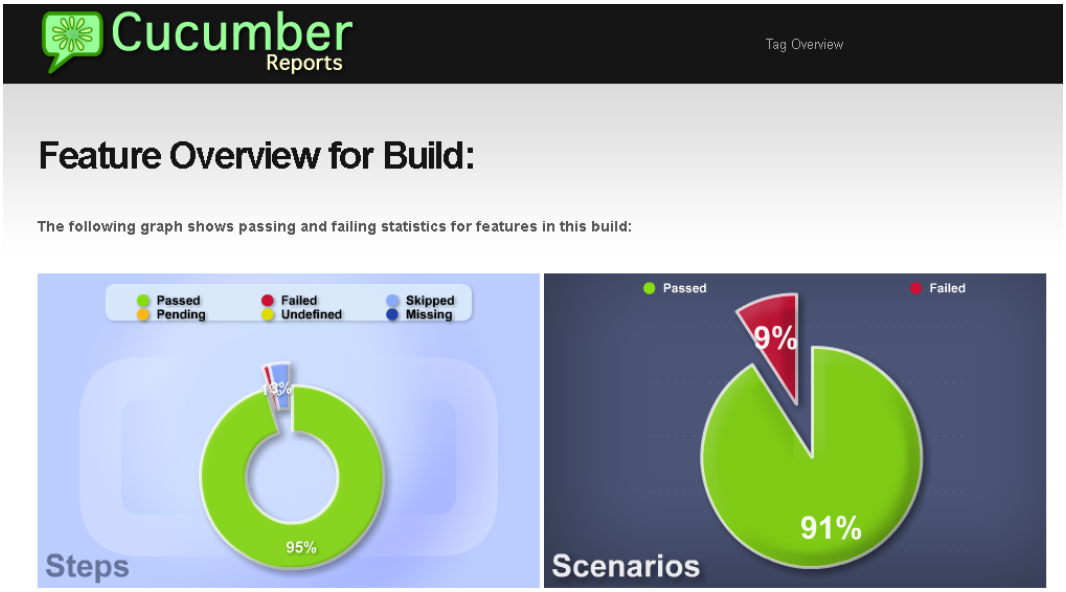
a. Feature
b. Total of no scenarios in each feature
c. Status
d. Duration

Feature in green color are successfully executed steps.

Those are in blue are skipped

When a step is failed, it will be shown in red color. Then next steps in the same feature will be skipped and shown in blue color.

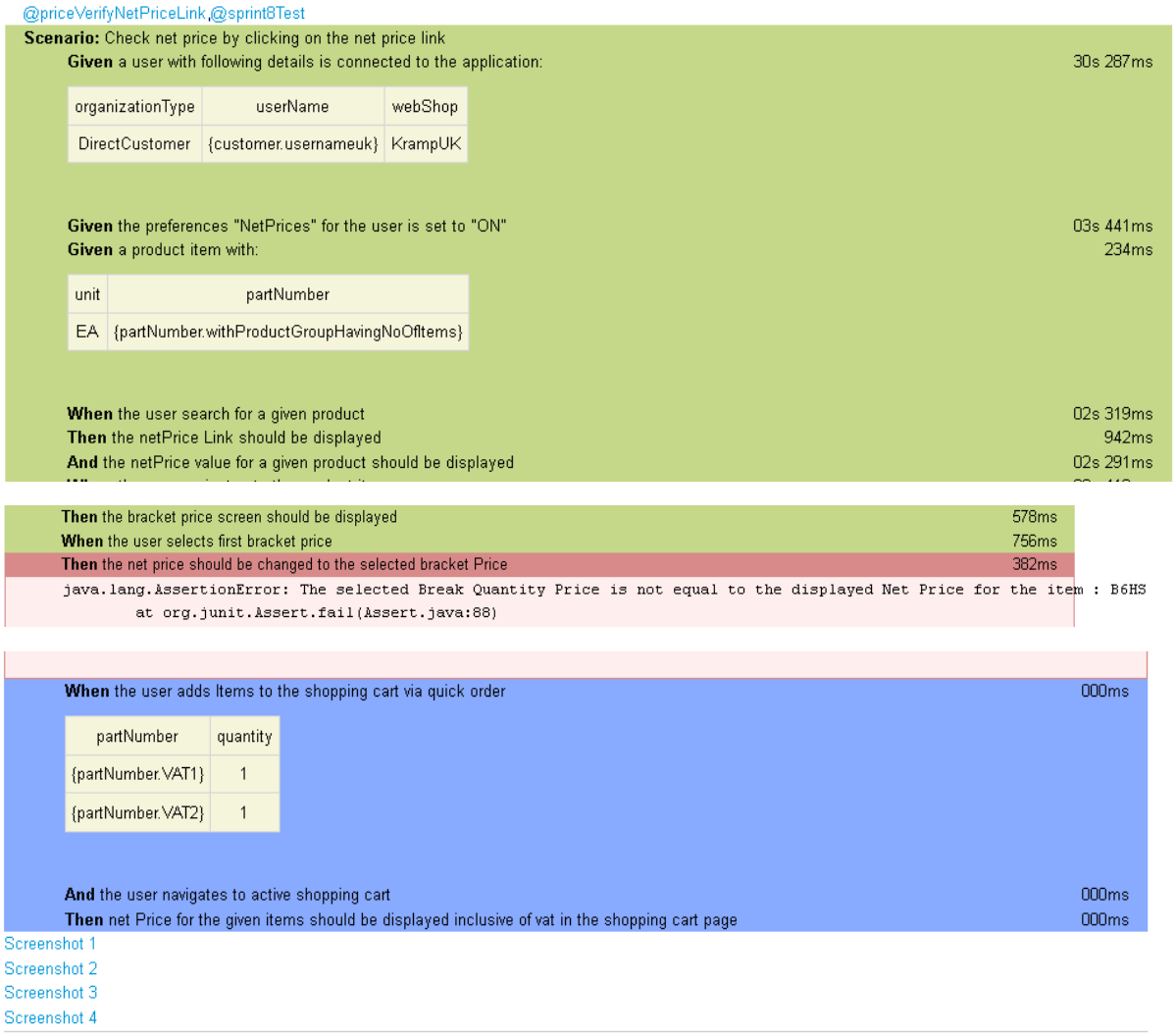
Below is a sample Cucumber Report.



Feature Statistics

	Scenarios			Steps								
Feature	Total	Passed	Failed	Total	Passed	Failed	Skipped	Pending	Undefined	Missing	Duration	Status
Pricing Functionality	11	10	1	87	83	1	3	0	0	0	09m 29s 527ms	failed
1	11	10	1	87	83	1	3	0	0	0	09m 29s 527ms	Totals

Below report shows the successfully executed, failed and skipped steps:



@priceVerifyNetPriceLink,@sprint8Test

Scenario: Check net price by clicking on the net price link

Given a user with following details is connected to the application: 30s 287ms

organizationType	userName	webShop
DirectCustomer	{customer.usernameuk}	KrampUK

Given the preferences "NetPrices" for the user is set to "ON" 03s 441ms

Given a product item with: 234ms

unit	partNumber
EA	{partNumber.withProductGroupHavingNoOfItems}

When the user search for a given product 02s 319ms

Then the netPrice Link should be displayed 942ms

And the netPrice value for a given product should be displayed 02s 291ms

Then the bracket price screen should be displayed 578ms

When the user selects first bracket price 756ms

Then the net price should be changed to the selected bracket Price 382ms

```
java.lang.AssertionError: The selected Break Quantity Price is not equal to the displayed Net Price for the item : B6HS
        at org.junit.Assert.fail(Assert.java:88)
```

When the user adds Items to the shopping cart via quick order 000ms

partNumber	quantity
{partNumber.VAT1}	1
{partNumber.VAT2}	1

And the user navigates to active shopping cart 000ms

Then net Price for the given items should be displayed inclusive of vat in the shopping cart page 000ms

Screenshot 1

Screenshot 2

Screenshot 3

Screenshot 4

7. Page Object Pattern

A Page Object Model is a design pattern that can be implemented using selenium WebDriver. It essentially models the pages/screen of the application as objects called Page Objects, all the functions that can be performed in the specific page are encapsulated in the page object of that screen. In this way any change made in the UI will only affect that screens page object class thus abstracting the changes from the test classes.

Advantage of page object pattern:

- Increases code reusability - code to work with events of a page is written only once and used in different test cases.
- Improves code maintainability - any UI change leads to updating the code in page object classes only leaving the test classes
- Makes code more readable and less brittle

Implementing POM (Page Object Model) with WebShop Automation:

Base Page:

Every page class should inherit the Base Page. This is implemented in Selenium Wrapper. Methods to initialize all the page objects are implemented in this class.

Declaration of a page class looks as below:

```
@Component
public class MyAccountPage extends BasePage{

	private static final Logger LOGGER = Logger.getLogger(SalesCatalogPage.class);

	@Autowired
	Environment properties;

	@Autowired
	WebShopHomePage webShopHomePage;

	@FindElement(page = "Login", field = "myAccountIcon")
	public Button myAccountIcon;

	@FindElement(page = "MyAccount", field = "myAccountLink")
	public Link myAccountLink;
```

The Page Class shown above is annotated with “@Component”. This spring annotation enables the framework to initialize and keep the object in the Application Context. Hence the page object can be accessed across the step definitions.

All the objects in a page class should be declared with annotation:

- @FindElement
- @FindElements

Example: myAccountIcon in Login page. In the above example it is declared as @FindElement (page = “Login”, field = “myAccountIcon”)

Some of the methods in the MyAccountPage are as below:

- public void navigateToPreferences()
- public void selectOrderHistory()
- public void enableItemPreview()

As per the design strategy of Page Object Pattern, these methods represent the user actions on the page.

8. Object Repository

Objects on a web page are identified by the Selenium using the below locators:

- Id
- Name
- Link Text
- Partial Link Text
- Tag Name
- Class Name
- CSS
- XPATH

Example:

```
WebElement element = driver.findElement(By.name("logonId"));
  //click on the object
element.click();
```

Here in the above example, the “logonId” is hardcoded in the Java code. To avoid this, the webshop automation framework provides a way to store all the locators in a centralized place called “Object Repository”. This is an XML file and the structure of which is as below:

```
<?xml version="1.0" encoding="UTF-8"?>
<objectRepository>
    <page name="Login">
                <field name="loginIcon" findBy="xpath" findByValue="//a[@class='header-btn btn-login']"/>
                <field name="username" findBy="name" findByValue="logonId"/>
                <field name="password" findBy="id" findByValue="logonPassword"/>
                <field name="loginButton"  findBy="name" findByValue="logOnButton"/>
                <field name="myAccountIcon"  findBy="xpath" findByValue="//a[@class='header-btn btn-profile']"/>
                <field name="myAccountUserName"  findBy="xpath" findByValue="//table[@class='table unstyled']//tr[1]//td[2]"/>
                <field name="logout"  findBy="xpath" findByValue="//div[@class='profileLayerBottom']//li[last()]//a"/>
                <field name="krampLogo" findBy="xpath" findByValue="//a[@id='logo']"/>
                <field name="krampHeader" findBy="xpath" findByValue="//ul[@id='nav']"/>
                <field name="closeSurvey" findBy="xpath" findByValue="//div[@id='surveyModal']//button[@id='closeSurvey']"/>
        </page>
</objectRepository>
```

“<objectRepository>” is the root element and inside this we have “<page name=”[name]”> tag. This holds all the field elements. Each field element has the attributes, “name”, “findBy”, and “findByBalue”. The object repository is read by the selenium web driver wrapper while initializing all the pages before starting the test execution.

Every object in a web page is declared as below in each page class:

```
@FindElement(page = "MyAccount", field = "returnHistoryDetails")
public Link returnHistoryDetails;
```

Field =”[field name]” refers to the attribute “name” in the “field” element of the object repository and page=”[page name]” refers to the attribute “name” of the “page” element.

If the page name and field name are not provided with @FindElement OR @FindElements annotations then the name of the Page class is taken as the page name and variable name is taken as the field name. For example:

@FindElement

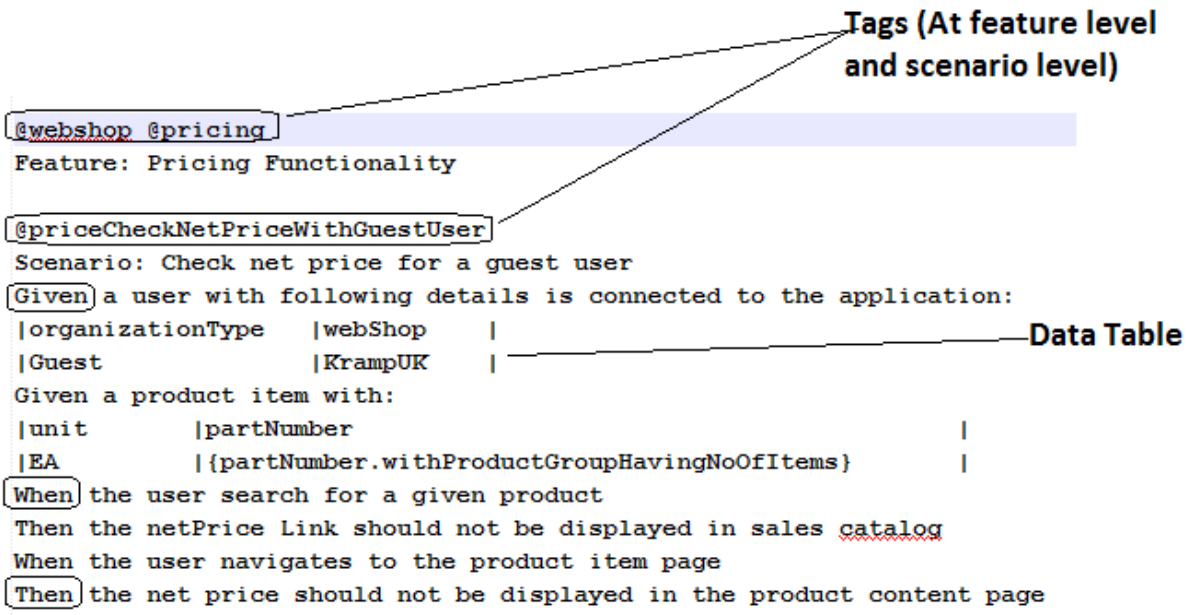
Public TextBox loginUserName;

9. Writing Feature Files

Feature files are written in domain specific language called “Gherkin”. Gherkin is a domain specific language used in Behavior Driven Development (BDD) for writing acceptance tests in the form of feature files. Feature files are written in Given-When-Then format. The main keywords used for writing feature file are:

- Feature
- Scenario
- Given, When, Then, And
- Scenario Outline
- Background
- Examples

Example of a feature file:



In the above feature file example usage of various Gherkin keywords can be seen.

A feature file (.feature) contains scenarios related to one particular feature of the system. Precondition to execute a particular scenario is specified in "**Given**" step. User's action is specified in the "**When**" step. Result of the scenario execution is verified in "**Then**" step. Tag specified at the top of the feature file can be used to execute all the scenarios in the feature file together. Tag specified at the scenario level can be used to execute a particular test scenario.

Writing a sample feature file:

```
Feature: Calculator

@addTwoNumbers
Scenario: Add two numbers
        Given I have entered 30 into the calculator
        And I have also entered 50 into the calculator
        When I press add
        Then the result should be 80 on the screen
```

In this example, user will enter numbers 30 and 50 into the calculator. When he presses add button, it should display 80 (Then is the verification step).

Data Tables

Data tables are used to pass a list of values to a step.

```
Given a user with following details is connected to the application:
|organizationType   |webShop    |
|Guest              |KrampUK    |
```

To login to the application as a random user several parameters are required to get the user from the database. These parameters are passed a list from the data table.

Scenario Outline

If there are several scenarios that differ only by their input data then the “Scenario outline” with “Examples” can be used.

```
@quickAddForReturnableNotRutunrable&AdvisoryProducts @deleteActiveShoppingCart
Scenario Outline: Quick add in Shopping cart
Given a user with following details is connected to the application:
|organizationType    |webShop      |
|DirectCustomer     |KrampUK    |

When the user creates a "shared" shopping cart with "cartForTesting"
When the user adds the following items to shopping cart via quick add
|partNumber     |quantity   |   comments    |
|<partNumber>   |       1   |   FirstComment|
Then product "<partNumber>" should be added to shopping cart with quantity "1" and comment "FirstComment"
When the user adds the following items to shopping cart via quick add
|partNumber     |quantity|  comments        |
|<partNumber>   |       1|  SecondComment   |
Then product "<partNumber>" should be added to shopping cart with quantity "2" and comment "SecondComment FirstComment"

Examples:
|productType    |    partNumber            |
|returnable   | {partNumber.returnable}  |
|notreturnable| {partNumber.notreturnable}  |
|advisory| {partNumber.advisory}        |
```

In the above example, the scenario will be executed for different types of products. Only the different sets of data is used to execute the scenario.

10. Developing Glue code / Step definition

When Cucumber executes a Step in a Scenario it will look for a matching Step Definition to execute. These Step Definitions are written in Step File using languages like Java, Ruby and Python etc. The Java method is developed for each step written in the feature file.

A Step Definition is a small piece of code with a (regex) pattern attached to it. The pattern is used to link the step definition to all the matching Steps and Cucumber will execute the code written inside Step Definition when it sees a Gherkin Step.

To understand how Step Definitions work, consider the following Step File which is written against the Feature File below.

```
@Given("^the user navigates back to the previous product group using breadcrumb$")
public void navigateBackToProductGroupFromBreadCrtumb() throws  AutomationElementNotFoundException{
    categoryNameToNavigate = salesCatalogPage.getLastCategoryText();
    salesCatalogPage.navigateBackToCategoryUsingBreadCrumb();
}
```

Step in the Feature file:

```
Scenario Outline: Check net price by clicking on the net price link
Given a user with following details is connected to the application:
|organizationType       |webShop    |
|<organizationType>     |<webShop>  |
When the user searches for a product item with more than 1 product group
Then the search results should be displayed with the default product group for the product item
When the user drills down to the lowest category level with the other product group from category pane
Then the product group displayed should be in acordance with the category chosen
When the user clicks on the product item
Then the breadcrumb should show the path to the right product group
When the user navigates back to the previous product group using breadcrumb
Then the product group displayed should be in accordance with the breadcrumb

Examples:
|organizationType   |webShop    |
|DirectCustomer     |KrampUK    |
|Guest              |KrampUK    |
```

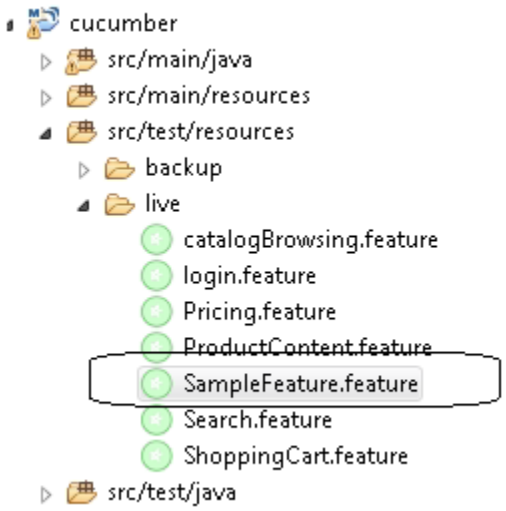
11. Creating a Sample Test

In previous sections, Test Automation, Cucumber, etc are explained in detail. As a final step, we will look into creating a sample test using the framework here.

As an example, we will consider a simple scenario:

```
@webshop @search
Feature: Search Functionality

@searchProduct
Scenario: Search for a product on Kramp UK Webshop
Given a guest user navigates to Kramp UK webShop
And searches for the product "100100"
Then the product "100100" is displayed in search results
```

Create feature file in “src/test/resources”:



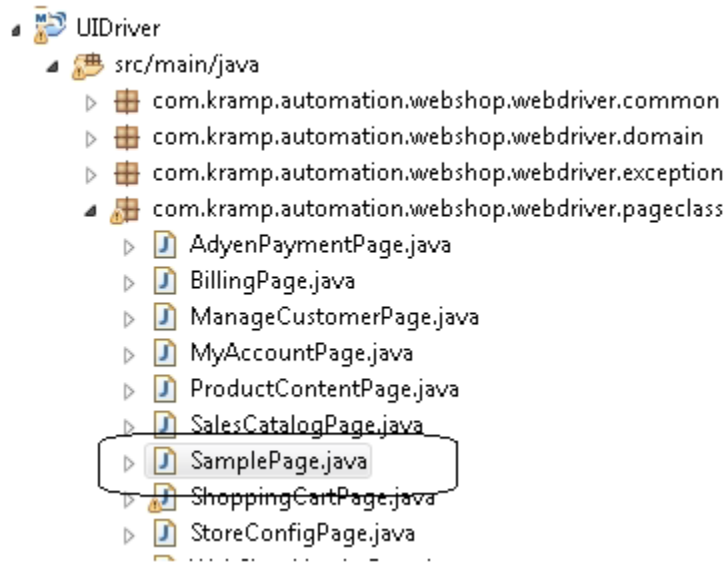
Enter the below steps into the feature file.

```
@webshop @search
Feature: Search Functionality

@searchProduct
Scenario: Search for a product on Kramp UK Webshop
Given a guest user navigates to Kramp UK webShop
When searches for the product "100100"
Then the product "100100" is displayed in search results
```

Create entries into the “ObjectRepository.xml” in “src/main/resources” as shown below:

```
<page name="SampleSearch">
    <field name="searchBox" findBy="name" findByValue="query" />
    <field name="searchButton" findBy="xpath" findByValue="//button[@class='btn btn-search btn-warning']/i" />
    <field name="searchResultsPartNumber" findBy="xpath" findByValue="//li[@data-part-number='+[PartNumber]+']//p[@class='partNumber']" />
</page>
```

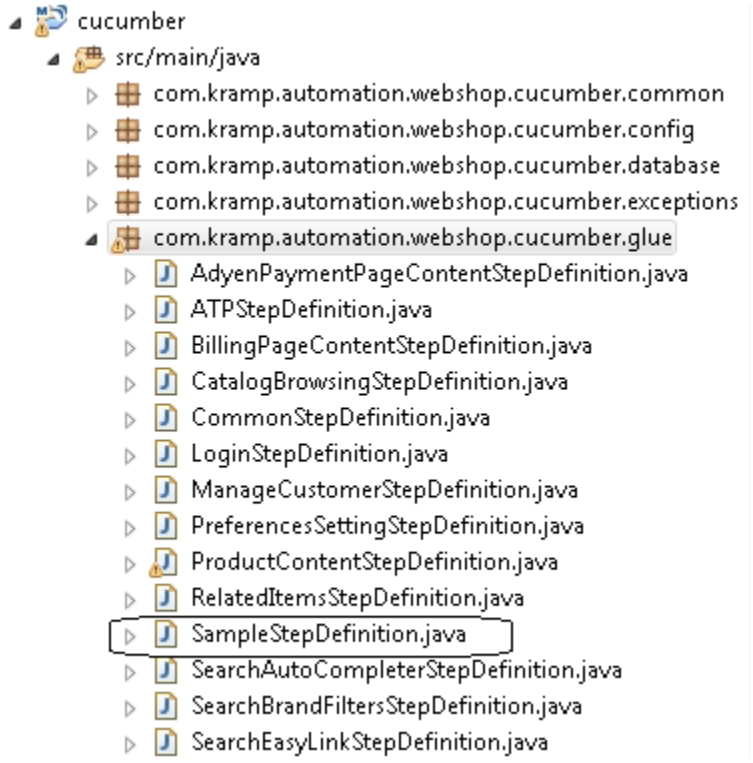
Create a Page Class “SamplePage.java” in “com.kramp.automation.webshop.webdriver.pageclass” package. This package is under **UIDriver** project as this is separated as a different layer in the framework.



Sample Page class should have the below code:

```
package com.kramp.automation.webshop.webdriver.pageclass;

import org.apache.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import com.kramp.automation.framework.selenium.exceptions.AutomationDriverException;
import com.kramp.automation.framework.selenium.exceptions.AutomationElementNotFoundException;
import com.kramp.automation.framework.selenium.fields.Link;
import com.kramp.automation.framework.selenium.fields.TextBox;
import com.kramp.automation.framework.selenium.pageinitializers.BasePage;
import com.kramp.automation.framework.selenium.utilities.FindElement;
import com.kramp.automation.framework.selenium.webdriver.SeleniumWebDriver;
import com.kramp.automation.webshop.webdriver.common.Utilities;

@Component
public class SamplePage extends BasePage {

    private static final Logger LOGGER = Logger.getLogger(ProductContentPage.class);

    @Autowired
    Utilities utilities;

    @FindElement(page = "SampleSearch", field = "searchBox")
    public TextBox searchBox;

    @FindElement(page = "SampleSearch", field = "searchButton")
    public Link searchButton;

    @FindElement(page = "SampleSearch", field = "searchResultsPartNumber")
    public Link searchResultsPartNumber;


    @Autowired
    public SamplePage(SeleniumWebDriver driver) throws AutomationDriverException {
        super(driver);
    }

    public void searchProduct(String partNumber) throws AutomationElementNotFoundException{
        LOGGER.info("Entering the part number "+partNumber+"into the search box");
        searchBox.typeText(partNumber);
        searchButton.click();
        utilities.waitForJSAndJQueryToLoad();

    }

    public boolean isProductPresentInSearchResults(String partNumber){
        searchResultsPartNumber.replaceSubStringOfFindByValue("+[PartNumber]+", partNumber);
        LOGGER.info("Checking whether the product "+partNumber+"is present in the search results");
        return searchResultsPartNumber.isPresent();
    }

}
```

Now the step definition for the above feature should be created in “com.kramp.automation.webshop.cucumber.glue” as “SampleStepDefinition.java”.



The step definition java class should contain the below code.

```
package com.kramp.automation.webshop.cucumber.glue;

import org.apache.log4j.Logger;
import org.junit.Assert;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.env.Environment;
import com.kramp.automation.framework.selenium.exceptions.AutomationDriverException;
import com.kramp.automation.framework.selenium.exceptions.AutomationElementNotFoundException;
import com.kramp.automation.framework.selenium.webdriver.SeleniumWebDriver;
import com.kramp.automation.webshop.webdriver.common.Utilities;
import com.kramp.automation.webshop.webdriver.pageclass.SamplePage;
import com.kramp.automation.webshop.webdriver.pageclass.WebShopHomePage;
import cucumber.api.java.en.Given;
import cucumber.api.java.en.Then;
import cucumber.api.java.en.When;

public class SampleStepDefinition {

    @Autowired
    Environment properties;

    @Autowired
    SeleniumWebDriver webDriver;

    @Autowired
    SamplePage samplePage;

    @Autowired
    WebShopHomePage webShopHomePage;

    @Autowired
    Utilities utilities;

    private static final Logger LOGGER = Logger.getLogger(ManageCustomerStepDefinition.class);

    @Given("^a guest user navigates to Kramp UK webShop$")
    public void guestUserNavigatestoKrampUK() throws AutomationDriverException{
        LOGGER.info("Launching kramp UK webshop");
        webShopHomePage.launchUrl(properties.getProperty("environment.webshop.url") + "do/action/shop-gb/en");
        utilities.waitForJSAndJQueryToLoad();
        webDriver.captureScreenshot();
    }

    @When("^searches for the product \"(.*?)\"$")
    public void searchesGivenProduct(String partNumber) throws AutomationDriverException,
                                                        AutomationElementNotFoundException{
        LOGGER.info("Searching for the product "+partNumber);
        samplePage.searchProduct(partNumber);
        utilities.waitForJSAndJQueryToLoad();
        webDriver.captureScreenshot();
    }


    @Then("^the product \"(.*?)\" is displayed in search results $")
    public void verifyWhetherTheProductIsDisplayed(String partNumber) throws AutomationElementNotFoundException{
        LOGGER.info("Verifying the product "+partNumber+ " is present in Search Results page");
        Assert.assertTrue("The product being searched is not present: "+partNumber,
                samplePage.isProductPresentInSearchResults(partNumber));
    }
}
```

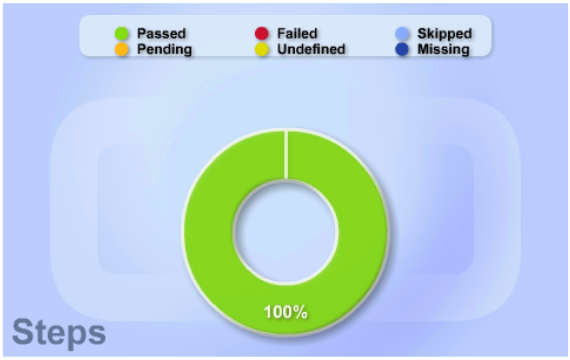
Execute the Sample test as explained in Test Suite Execution which generates the below Cucumber Report.

Cucumber Reports

Tag Overview

Feature Overview for Build:

The following graph shows passing and failing statistics for features in this build:



Feature Statistics

	Scenarios			Steps								
Feature	Total	Passed	Failed	Total	Passed	Failed	Skipped	Pending	Undefined	Missing	Duration	Status
Search Functionality	1	1	0	3	3	0	0	0	0	0	07s 930ms	passed
1	**1**	**1**	**0**	**3**	**3**	**0**	**0**	**0**	**0**	**0**	**07s 930ms**	**Totals**

	Scenarios			Steps								
Feature	Total	Passed	Failed	Total	Passed	Failed	Skipped	Pending	Undefined	Missing	Duration	Status
Search Functionality	1	1	0	3	3	0	0	0	0	0	07s 930ms	passed

@webshop,@search

Feature: Search Functionality

@searchProductSampleTest

Scenario: Search for a product on Kramp UK Webshop

Given a guest user navigates to Kramp UK webShop	04s 564ms
When searches for the product "100100"	03s 269ms
Then the product "100100" is displayed in search results	097ms

Appendix

Resources:

[1]. https://github.com/cucumber/cucumber/wiki/Feature-Introduction

[2]. "Manage time and resources better by scheduling automated tests" (Vaibhav Rangare, developerWorks, June 2011): Take a higher level view of automated testing, with this introduction to planning an efficient and maintainable test suite.

[3]. Selenium: Page Objects (Selenium Wiki): Learn more about using the Page Object pattern for test automation.

[4]. https://en.wikipedia.org/wiki/Test_automation#Framework_approach_in_automation

[5]. http://www.srccodes.com/p/article/48/cucumber-test-behavior-driven-development-bdd-feature-step-runner-glue-gherkin-data-table-scenario-given-when-then