Part 2: A Conceptual Test Automation Framework Design Using A Hybrid Approach

By Obi Nzeadibe

In Part 1, I discussed a general outline of the Page Object Model Design approach for automating the testing of software applications. Taking it a step further, I will now demonstrate how to use a hybrid approach that includes other design patterns like Data Driven Design and Behavior Driven Design in addition to POM to create an automation framework. I will automate two web pages of http://www.wordpress.com to demonstrate how the hybrid approach works. The complete complete code will be uploaded to my github account and can be downloaded from https://github.com/obimann1/Wordpress

Keywords: Behavior Driven Development, Gherkin, Eclipse, Custom Report Generation,Scenario outline, Code Optimization

It is important to describe some of the key concepts in the proposed hybrid design. Cucumber is one of the software tools that supports Behavior driven development(BDD).It runs automated tests written in BDD style. BDD outlines application behavior in simple English defined by a language called Gherkin. The major advantage of Gherkin is the ability to write automation scripts in plain English text that provides a bridge between business and technical language. For ease of understanding, an example of a cucumber feature file describing the behavior of two scenarios to be automated shows the various keywords used in Gherkin.

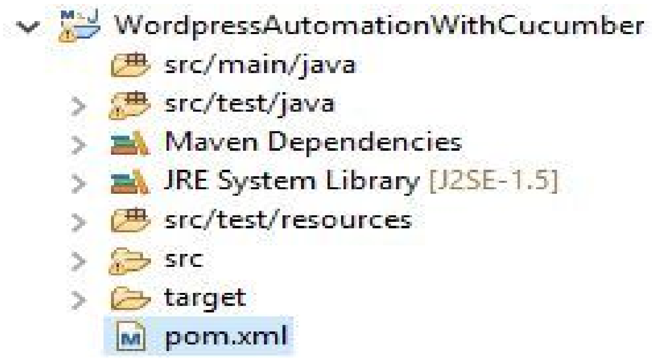
```
Feature: Test Login Functionality

Scenario: Login test with valid credentials
        Given open the wordpress website
        When Click on Login Link
        When Enter user name “Martin”
        When Enter password “Martin@123”
        When Click on Login button
        Then Login should succeed
        Then Close browser

Scenario: Test Login With Invalid Credentials
        Given Open the wordpress website
        When Click On Login Link
        When Enter user name  “abc”
        When Enter password “abc123”
        When Click on Login Button
        Then Error message should be displayed
```

Other Gherkin keywords not used in the above example feature file are: Background, And, But, Scenario Outline ,Examples and * (used to replace any keyword).

Step 1: Create a Maven project on Eclipse IDE
And add the following dependencies from the maven repository to the pom.xml of your project: selenium-java, cucumber-java, cucumber-junit and junit dependencies. For custom report generation, we will need to add additional plugins like the maven compiler plugin, maven surefire and the cucumber reporting plugin. With the dev environment now properly configured, the next step will be to set out the structure of the proposed project on Eclipse.

Step 2: Maven Project Layout on Eclipse

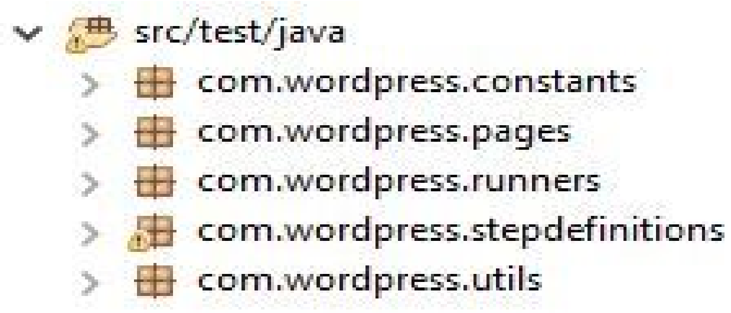
Fig 1: Default Project Folder Structure


WordpressAutomationWithCucumber
src/main/java
src/test/java
Maven Dependencies
JRE System Library [J2SE-1.5]
src/test/resources
src
target
pom.xml

Step 3: Create All the Packages and Folders/Files

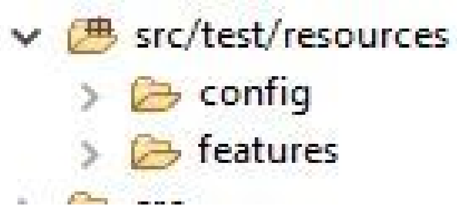
In the **src/test/java folder** we will create the following packages:

Fig 2: Required Packages


src/test/java
com.wordpress.constants
com.wordpress.pages
com.wordpress.runners
com.wordpress.stepdefinitions
com.wordpress.utils

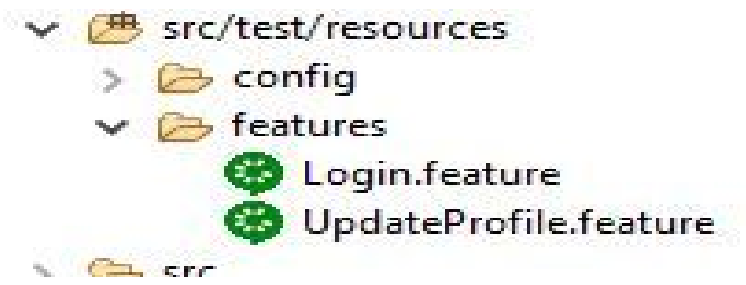
In **src/test/resources**, we will create 2 additional folders as shown below:

Fig 3: Resource folders


src/test/resources
config
features

Step 4: **Feature files**
With the conceptual structure of the framework properly laid out as shown above, the next step is to translate the Application/Business Requirements into test cases. This is done by creating Feature files that describe one or more scenarios to be tested. We will be using the Login and Profile pages of the wordpress website as illustration. This two feature files are placed in the **src/test/resources/features** folder:

Fig 4: Feature files for the 2 test pages



As stated earlier on, a feature file contains a set of scenarios or test cases. Each scenario contains a set of steps starting with a Gherkin keyword. Each step of a scenario will eventually map to a step definition. It is important to observe that both feature files contain steps that are similar. To avoid code duplication, the background keyword is introduced in the feature file. Background in Cucumber is used to define a step or series of steps which are common to all the tests in the feature file. A Background runs before each and every scenario in the feature file in which it is defined.

Scenario Outline

Scenario outline is used to achieve data driven concept in Selenium.The data driven concept is used in situations where multiple scenarios are having the same steps but differ only in data, then we can use data driven concept for script development.

```
Scenario: Test Login with invalid credentials
    Given open browser and load the wordpress
    When click on login link
    When enter user name "abc"
    When enter password "abc123"
    When click on login button
    Then error message should be displayed

Scenario: Test Login with invalid credentials using numbers
    Given open browser and load the wordpress
    When click on login link
    When enter user name "7594375934"
    When enter password "894358943"
    When click on login button
    Then error message should be displayed


Scenario Outline: Test login with invalid credentials
        Given open browser and load the wordpress
        When click on login link
        When enter user name "<username>"
        When enter password "<passowrd>"
        When click on login button
        Then error message should be displayed

        Examples:
        |       username   |          password   |
        |       abc        |          abc123     |
        |       7594375934 |          894358943  |
```

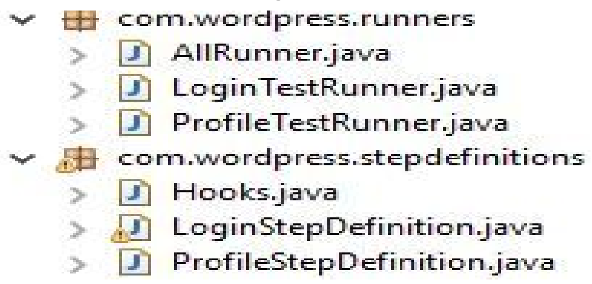
Step 5: Step Definition Files
The Step definition files contains a block of code that is executed when any step of the scenario in a feature file is executed. Every step in a scenario must have a corresponding step definition. For example,

Given open wordpress application
When click on login link.

Will have a corresponding code on the Step Definition file that opens the wordpress website and clicks on the login link.

A runner class is used to execute the feature file and generate the step definition code. We will therefore create the corresponding Step Definition files and their runner classes in the packages com.wordpress.stepdefinitions and com.wordpress.runners as shown below.

Fig 5: Step Definition Files/Runner Classes



```
com.wordpress.runners
    AllRunner.java
    LoginTestRunner.java
    ProfileTestRunner.java
com.wordpress.stepdefinitions
    Hooks.java
    LoginStepDefinition.java
    ProfileStepDefinition.java
```

The LoginTestRunner.java class runs the Login.feature file and generates snippet of the LoginStepDefinition.java code. It is usually executed as a Junit file. You can run a separate testrunner class for each corresponding feature file or use one runner class to operate on all the feature files.

Fig 6: LoginTestRunner.java

```
package com.wordpress.runners;
import org.junit.runner.RunWith;
import cucumber.api.CucumberOptions;
import cucumber.api.junit.Cucumber;
@RunWith(Cucumber.class)
@CucumberOptions(
            features={"src/test/resources/features/Login.feature"},
            glue= {"com.wordpress.stepdefinitions"},
            plugin ={ "pretty" , "html:target/report/cucumber",
            "json:target/report/json/login.json"})
      public class LoginTestRunner {
      }
```

Fig 8: Part Snippet of the LoginStepDefinition.java

```
@Given("User open Browser and load the URL")
public void user_open_Browser_and_load_the_URL() {
  // Write code here that turns the phrase above into concrete actions
  throw new PendingException();
}
```

```
@Then("home page should appear")
public void home_page_should_appear() {
   // Write code here that turns the phrase above into concrete actions
   throw new PendingException();
}
```

Step 6: Create Page Object classes

The Login and Profile web pages are represented as java classes, and the various elements on the page are defined as variables of the class. All possible user interactions can then be implemented as methods on the class. The page classes are placed within the package shown in Fig 9 and the code can be downloaded from Github WordPress Project under my username “obimann”.

Fig 7: Page classes

The Page Factory
Page factory is an extension to the page object model. It is a list of all the locators of a particular page of the Application Under Test (AUT) and is used to initialize elements of the Page Object or instantiate the Page Objects itself.

- Example:-

```
@findBy(how=How.ID, using="username")

WebElement userNameTextElement;


Public void setUserName(String username)

        {UserNameTextElement.sendKeys(username);}
```

- Initiating Page Class:-

```
LoginPage lPage = PageFactory.initElements(driver, LoginPage.class);
```

Using the Page factory approach helps with code optimization. Ordinarily the code for inputting username would require 3 lines of code like:

```
WebElement element = driver.findElement(By.id(“username”)) ;
element.driver.clear();
element.sendKeys(“username”);
```

But with optimization, it is reduced to:

```
@When("enter user name {string}")
public void set_user_name(String userName) {
        lPage.setUserName(userName);}
```

Step 7: Property files

It is not a good coding practice to store store hard coded values in our automation script. These values can be eliminated from our script by use of a property file. In this file, we store project configuration data, database config or project settings . Each parameter in properties file are stored as a pair of strings, in key and value format, where each key is on one line. You can easily read properties from some file using object of type Properties. A sample property file will be like:

```
Email    =     //*[@id='Email']
Pwd      =     //*[@id='Passwd']
Submit   =     //*[@id='signIn']
```

Rather than the usual method of finding locators, we will simply read in the property using the key value concept in java.

```
FileInputStream objfile = new
FileInputStream(System.getProperty("./src/test/resources/config/config.properties");
obj.load(objfile);
        driver.get("https://www.hotmail.com/");
        driver.findElement(By.xpath(obj.getProperty("Email"))).click();
        driver.findElement(By.xpath(obj.getProperty("Pwd"))).click();
        driver.findElement(By.xpath(obj.getProperty("Submit"))).click();
```

Step 7: Define Tags

An Application Under Test will consist of many Feature files designed to test different aspects of the Application. Each Feature may require one or more scenarios to test it. The different scenarios aim to test different behaviors of the application feature for different purposes such as UAT (User Acceptance Testing) for the Client, Smoke test, New Feature Test, Regression/Integration test for the Developer, Quality Assurance Engineers or Business Analysts. In Cucumber, we use Tags in the Feature File to organize scenario execution. Each scenario can be defined with one or more tags.The tags so defined will subsequently be specified in the runner class. So at runtime, only the specified tags are executed. Tag starts with "@". After "@" you can have any relevant text to define your tag. Examples of tags are @smoketest(for scenarios to be executed for smoke testing to determine the stability of the build), @regression (for testing features during regression), @sanity(for testing critical features before final release to customer). There is no limit to the number of tags that can be used in Feature files. Some tags can also be defined at a feature level. Once you define a tag at the feature level, it ensures that all the scenarios within that feature file inherits that tag. A Scenario or feature can have as many tags as you like. Just separate them with spaces as this: @billing @bicker @annoy. For multiple tags, we can define the logical and/or operation in the JUnit runner class as shown below:

Logical or (in runner class)	"**@smoketest,@regression"** means to execute scenarios matching @smoketest as well as @regression
Logical and (in runner)	**"@dev", "@sanity"** means to execute all scenarios matching both @dev and @sanity together.

Step 8: Hooks And Scenario Background

Hooks are blocks of code that can run at various points in the Cucumber execution cycle.There are two types of hooks:

- Before (executed before every scenario)
- After (After hook is executed after every scenario)

```
@Before
public void doSomethingBefore()

{//for example, code to create driver object}

@After
public void doSomethingAfter(Scenario scenario)

{ // for example, code to close browser}
```

Cucumber provides the **Background** keyword for steps that are similar in several scenarios. The order of execution is **Before Hook -> Background -> Scenario**

Step 9: Parallel Execution

The major objective for test automation is for faster feedback to developers, engineers and Business Analysts. As test cases grow in number, it becomes difficult to get the desired quick feedback. A workaround to this difficulty is to have test cases execute in parallel rather than sequentially. That is to say that if an application is required to be tested on three different browsers like internet explorer, firefox and google chrome, the automation script will invoke all three browsers at the same time rather than consecutively from one browser to the other.This will greatly reduce total execution time for testing and achieve a greater test coverage.
There are several ways to attain this objective but what I have implemented in the code revolves around the use of the maven surefire plugin as a dependency in Maven pom.xml file. The parameter **`forkCount`** in the plugin defines the maximum number of JVM processes that maven-surefire-plugin will spawn *concurrently* to execute the tests.In the configuration you can see the "include" parameter which contains the property "**/*TestRunner.class". This means that any classes with a name which ends in "TestRunner.class" will be run as a test.

Step 10: Custom Report Generation

Cucumber does not have a fancy reporting mechanism by default. By specifying the pretty plugin in CucumberOptions of our runner file like this:

```
@CucumberOptions( plugin = { "pretty" } )
```

We can generate reports in HTML, XML, JSON & TXT formats individually as desired or in all formats at the same time with this configuration:

```
@CucumberOptions(
        features = "src/test/resources/features/",
        glue= {"stepDefinitions"},
        plugin = { "pretty", "json:target/cucumber-reports/Cucumber.json",
                   "junit:target/cucumber-reports/Cucumber.xml",
                   "html:target/cucumber-reports"},
        monochrome = true
```

JSON formatted report is great for integration with other tools (jenkins plugin, reports, etc). Pretty HTML format is OK during development phase for Developers and Quality Engineers. But Custom Cucumber-Reports is generally used for reporting outside of the dev team to managers. To configure this type of report requires the following dependencies to be added to our Maven POM file:

Maven Compiler Plugin
Maven Surefire Plugin
Maven Cucumber Reporting

I have modified the default configuration settings of the maven cucumber reporting plugin to:

```
<plugin>
                                          <groupId>net.masterthought</groupId>
                                           <artifactId>maven-cucumber-reporting</artifactId>
                                          <version>3.10.0</version>
                                          <executions>
                                           <execution>
                                           <id>execution</id>
                                           <phase>verify</phase>
                                          <goals>
                                           <goal>generate</goal>
                                          </goals>
                                          <configuration>
                                           <projectName>Wordpress-Consolidate-Report</projectName>
              <outputDirectory>${project.build.directory}/report/consolidate-reports</outputDirectory>
              <cucumberOutput>${project.build.directory}/report/json/</cucumberOutput>
              <enableFlashCharts>false</enableFlashCharts>
              <skippedFails>true</skippedFails>
             </configuration>
          </execution>
       </executions>
</plugin>
```

Obi Nzeadibe is a Software Automation Framework Design Expert based in North Delta, British Columbia. With over 18 years experience in Software Quality Assurance, he has worked with Xcert International Inc as Interoperability Analyst, RSA Security Inc as Intermediate Software Engineer and at Fortinet Technology Inc as Developer QA. He enjoys reading and writing.