SENG202 – Software Engineering Project Workshop

2020

Tutorial 7 – Acceptance Testing Using Cucumber

1. Introduction

Automated Acceptance tests - why we need them? Instead of a business stakeholder passing requirements to the development team without much opportunity for feedback, the developer and stakeholders collaborate to write automated tests that express the outcome that the stakeholder wants. We call them acceptance tests because they cover the requirements the software needs to meet in order to be accepted by the stakeholders. These are different from unit tests in the way that the goal of acceptance testing is to ensure that you are *building the correct product*, whereas unit tests are to ensure that you are *building the product correctly*.

In writing these acceptance tests, it is important to note that stakeholders may not be technically minded, therefore it is important to write acceptance tests in such a way that *both* the developers and the stakeholders understand their objectives. You've already seen this done with the "Given-When-Then" template, where the behaviour of the product including the inputs and expected outputs is described in a way that mutually understandable. However, it would be useful to perform these acceptance tests automatically to ensure the correct product is being built with each change. That is where cucumber comes in.

Cucumber

Cucumber is a toolkit based on behaviour driven development (BDD) and provides automation to the acceptance testing process. Cucumber takes a textual description of an acceptance test, which is written to the Gherkin[1] grammar specification, and checks to ensure your product performs according to the description. These descriptions are called *scenarios*. Each scenario is a set of steps for Cucumber to run through when testing the product and are stored in a `.feature` file. The feature file can contain multiple scenarios, as there can be multiple acceptance tests for a feature of your product.

The steps are written in plain text and need to be matched to step definitions, which define in code how Cucumber should interact with the product in order to execute the test. This glues the common language acceptance test to specific implementation and allows cucumber to evaluate the product. See Figure 1 for a diagram of how Cucumber works.

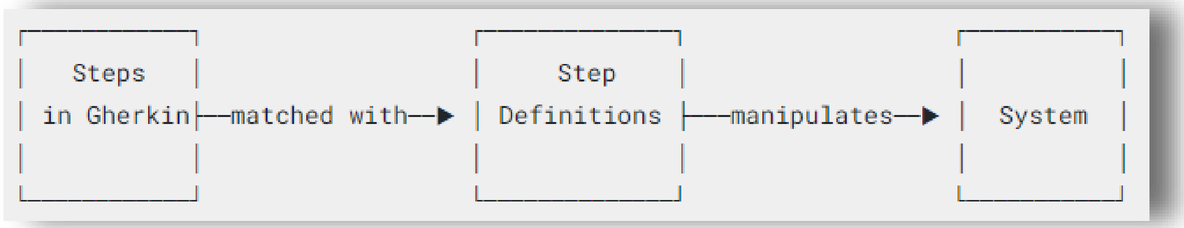

Figure 1: Basic diagram of how Cucumber works

[1] https://cucumber.io/docs/gherkin/

2. Instillation and Setup in IntelliJ

We will create a new Maven project and use that for this tutorial. Set up a new project with the command in the following snippet:

```
mvn archetype:generate                        \
   "-DarchetypeGroupId=io.cucumber"            \
   "-DarchetypeArtifactId=cucumber-archetype" \
   "-DarchetypeVersion=6.6.1"                \
   "-DgroupId=seng202"                   \
   "-DartifactId=cucumberlab"                \
   "-Dpackage=seng202"                   \
   "-Dversion=1.0.0-SNAPSHOT"                  \
   "-DinteractiveMode=false"
```

Snippet 1: Cucumber dependencies.

Open the generated folder in IntelliJ, you should find that IntelliJ imports the project as Maven as there is a pom.xml file that exists in that directory.

You may also want to specify the source and target compiler versions with the following snippet:

```
<properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
</properties>
```

Snippet 2: Compiler options.

Refresh the Maven project to apply the changes. You will see that the folder structure has already been set up with a test folder containing a resources folder and a java folder. The resources folder is where the **`.feature`** files are contained and the java folder contains two classes: **`StepDefinitions`** and **`RunCucumberTest`**. The first class tells Cucumber how to manipulate your product in order to execute the steps in the feature files. The second class tells Junit to use Cucumber to execute the tests as well as options regarding where the step definitions are located, as well as formatting for the test reports.

We recommend using the following options for the formatting:

```
plugin = {"pretty", "html:target/cucumber.html"}, snippets = CucumberOptions.SnippetType.CAMELCASE
```

Snippet 3: Cucumber options

If we run `**mvn test**` we will get a warning saying that Cucumber cannot find any feature files, lets create one with the following worked example.

3. Worked Example

Our goal today is to write a Java library that calculates the cost of your groceries at the supermarket. It will take two inputs: the prices of available items and the notification of items as they are scanned at the checkout. The checkout will keep track of the total cost. So, for example, if prices of available items looks like this:

Milk - $3.99

Bread - $1.99

When the following is scanned at the checkout:

1x Milk

Then the total cost will be $3.99. Similarly, if the following is scanned:

3x Bread

Then the total cost will be $5.97.

Define the Scenario

We want to create a scenario which describes an acceptance test for the above example. In order to do so we need to add a new feature file into resources/seng202 containing a the Gherkin description of the scenario. The following snippet describes a "blue skies" acceptance test for scanning one milk. Copy and paste this into a new file called "checkout.feature" in the above directory.

```
Feature: Checkout
Scenario: Scan 1 bottle of Milk
    Given the price of "Milk" is $3.99
    When I scan 1 "Milk"
    Then the total price should be $3.99
```

Snippet 4: Example scenario for the acceptance test of scanning 1 milk at the given price.

Running the Test and Implementing the Step Definitions

If we run our `` `mvn test` `` now with this feature we will find that the tests fail because there are undefined steps. The output also tells us some suggestions, in Java, for the step definitions. These code snippets are the glue that translates our textual test cases into executable code that actually manipulates the program and runs the methods. The below snippet contains the given suggestions. Note that the name milk and the numbers have been replaced with their data types which act as arguments for the definition functions and as variables for the step definitions.

```
@Given("the price of {string} is ${double}")
public void thePriceOfIs$(String string, Double double1) {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}
@When("I scan {int} {string}")
public void iCheckout(Integer int1, String string) {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}
@Then("the total price should be ${double}")
public void theTotalPriceShouldBe$(Double double1) {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}
```

Snippet 5: Suggestions for the step definitions.

Now we need to put these into a Java file. That is what the `StepDefinitions` class that was automatically generated is for. Copy your generated definitions into this file and rerun the test command. The tests will still fail however there is a nice output which shows each of the defined scenarios, the steps that produced errors as well as what the errors are.

Note: In your project you will have unit tests in the test folder as well as different files for the step definitions so it would be wise to put all the step definitions in their own package. When doing so we need to tell cucumber how to access them. To do this we use the “glue” option in the cucumber options in the **`RunCucumberTest`** class. For example, if all the step definitions are in test/java/seng202/steps then the options should look like the following snippet:

```
@CucumberOptions(glue="seng202/steps",
        plugin = {"pretty", "html:target/cucumber.html"},
        snippets = CucumberOptions.SnippetType.CAMELCASE)
```

Snippet 6: Options including glue

Now we need to actually implement the test steps so that they do something meaningful. For the first step we want to store the item price. When doing this we need to ensure that the price is accessible to the other step methods as the price will be used in the final assertion to ensure that the implementation being tested actually works. For the second step we want some method of adding the items price to the sale. An incomplete example is as follows:

```
public class StepDefinitions {
    private double price = 0;
    private Checkout checkout = new Checkout();

    @Given("the price of {string} is ${double}")
    public void thePriceOfIs$(String itemName, Double price) {
        this.price = price;
        //TODO: Support more than one price
    }
    @When("I scan {int} {string}")
    public void iCheckout(Integer quantity, String itemName) {
        checkout.add(quantity, this.price);
        //TODO: Support more than one price
    }
}
```

Snippet 7: Incomplete step definitions for this scenario

You may notice that these test definitions have a fatal flaw, and that is the fact that multiple different items cannot be added to the checkout because we have only defined one price variable. We will address this soon.

Now if we run the tests we will find that they are still failing, and that is because we haven’t implemented the **`Checkout`** class. Create a new class called Checkout in src/main/java/seng202 (You may need to create these directories yourself). Inside this class add the following method and attribute:

```
private double totalPrice = 0;

public void add(int quantity, double price) {
    this.totalPrice += (quantity * price);
}
public double getTotalPrice() {
    return this.totalPrice;
}
```

Snippet 8: Required functionality for the checkout class.

Rerun the tests, and you should now see that only the last step has failed as it is not yet implemented. This last step is what ensures that our product performs to the expected requirements.

Checking expected results

We will now write the final step, this is where we will put our Assert statements. An example of the final step is as follows:

```
@Then("the total price should be ${double}")
public void theTotalPriceShouldBe$(Double total) {
    Assert.assertEquals(total, checkout.getTotalPrice(), 0.001);
}
```

Snippet 9: The step definition to check the functionality is correct.

Note that there are three arguments for the assertEquals method, the first two are the variables to be compared and the last in the tolerance. We need a tolerance for float and double comparisons as computers are not 100% accurate when performing math computations on float/double variables.

Rerun the tests and now we see that all the steps for this scenario have passed. What we have just done is a part of BDD, in which we have taken an acceptance test which describes the way that the software should behave as a concrete example, and with this we have then developed the implementation for it. The acceptance test description must be negotiated with the stakeholders in order for BDD to be fully achieved.

Check in the target folder, there should be a file called **`cucumber.html`**. Open this in your browser and you should see a report for the scenarios that have been tested.

4. Exercise – Extend the above example

We mentioned that the given test descriptions would not be able to handle different items with different prices being added to the total. The task now is to go and modify the test step definitions so that the following scenario will be successfully run:

```
Feature: Checkout
Scenario: Scan 1 bottle of Milk and 1 bread
    Given the price of "Milk" is $3.99
    And the price of “Bread” is $1.99
    When I scan 1 “Milk”
    And I scan 2 “Bread”
    Then the total price should be $7.97
```

Snippet 10: New scenario to test

Do note, you do not need another step definition to handle the “And” parts of this scenario because we have already written the step definitions for the Given and the When steps. The only difference between two Given steps is the values given to the variables in the step definition (Milk vs Bread etc).

When modifying your test steps and the checkout implementation, consider the use of a model class “Item” and a HashMap<String, Integer> to store a list of the prices keyed by the name. If you need help please discuss with a tutor.

5. Further reading

There are many more features of Cucumber than what has been described here. Cucumber documentation is available here: https://cucumber.io/docs/cucumber/. This includes more worked examples with more complex scenario definitions as well as the API reference.

For more reading regarding behaviour driven development look here: https://dannorth.net/introducing-bdd/