SENG301 Tutorial - Acceptance Testing & Mocking

Fabian Gilson Marina Filipovic Patricia Inez de Andrade

16 Mar 2018, 10:24AM

Contents

1 Cucumber as a test automation tool

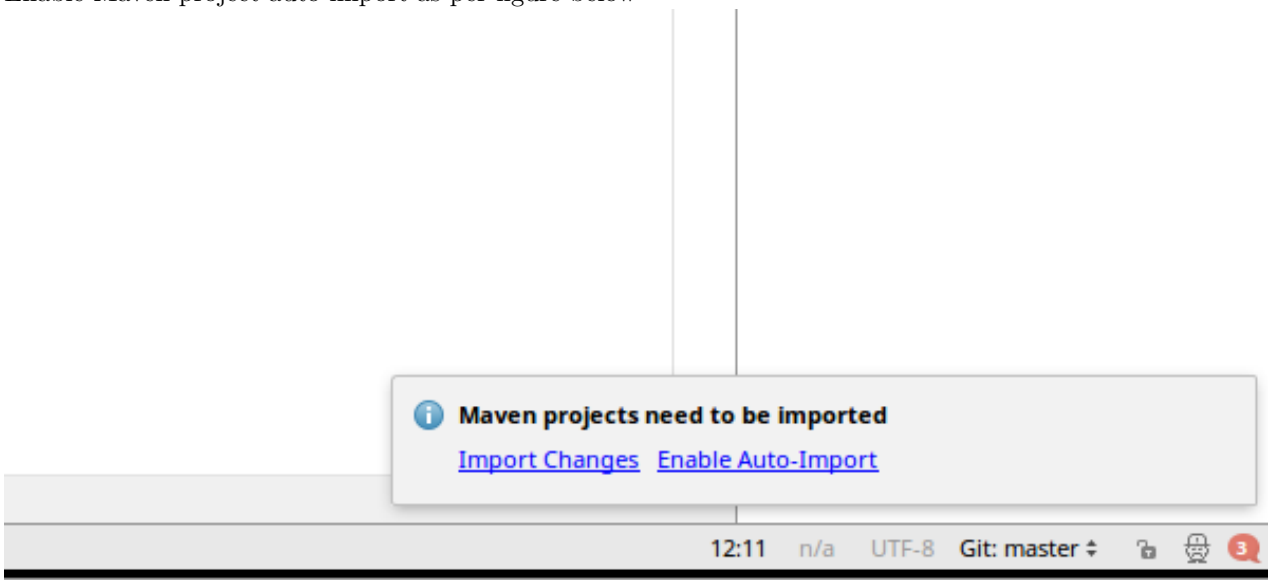
1.1 Introduction

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 stakeholder collaborate to write automated tests that express the outcome that the stakeholder wants. We call them acceptance tests because they express what the software needs to do in order for the stakeholder to find it acceptable. These tests are different from unit tests, which are aimed at developers and help them drive out and check their software design. It's sometimes said that unit tests ensure you build *the thing right*, whereas acceptance tests ensure you build *the right thing*.

- To be able to do TDD you need to work from outside-in, take customer's acceptance test that describes the behavior of the system from the customer's point of view.
- BDD (Behaviour-driven development builds upon test-driven development by formalizing the good habits of the best TDD practitioners. - write acceptance tests as examples that anyone on the team can read.
- Cucumber tests interact directly with the developers' code, but they are written in a medium and language that business stakeholders can understand. In this way these tests become a communication and collaboration tool.

1.2 Add Maven support to your existing WofInspection project (lab 3 project)

1. Right click on your project in IntelliJ and select "Add framework support...", and check the "Maven" technology.
2. Enable Maven project auto-import as per figure below



3. You will notice that you have a pom.xml file in your project now. To configure Maven to use the Java 8 language features and to compile classes with JVM 1.8, add the following lines to your pom file.

```
<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

Hint: Read more about it at: http://maven.apache.org/plugins/maven-compiler-plugin/examples/set-compiler-source-and-target.html

1.3 Set up Cucumber in IntelliJ

Set up cucumber dependencies in pom.xml file by adding the following with in dependencies tag:

```
<dependencies>

  <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>4.12</version>
       <scope>test</scope>
   </dependency>

   <dependency>
        <groupId>info.cukes</groupId>
        <artifactId>cucumber-java</artifactId>
        <version>1.2.5</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>info.cukes</groupId>
        <artifactId>cucumber-junit</artifactId>
        <version>1.2.5</version>
        <scope>test</scope>
    </dependency>

</dependencies>
```

1.4 Worked Example

1. Cucumber test are grouped into features 'features'. We use this name because we want them to describe the feature that a user will be able to enjoy when using our program. The first thing we need to do is make a folder (within our project) where we'll store features we'll be writing for our program. Create one under the Project name. We will store feature files in our feature folder. Let's create our first feature file called vehicle.feature. Right click the project and choose 'Run All Features in ....' Output will say that we do not have features defined at the moment and no Scenarios and no Steps.

2. Each Cucumber test is called a scenario, and each scenario contains steps that tell Cucumber what to do. This output means that Cucumber is happily scanning the feature directory, but it didn't find any scenarios to run. Let's create one. We will do this by editing feature file as follows:

```
   Feature: Update vehicle's model
  Scenario: Update a vehicle's model
 Given I am connected to the WOF database
   And The vehicle with plate "HVJ150" exists
When I update model of this vehicle to "Corolla"
Then Next time I retrive information on the vehicle, the vehicle model should be "
    ↪ Corolla"
```

3. Add **GenericTestRunner** class in your **test/java/** folder. This test will be calling Cucumber API to execute tests in our **features** folder. Add these imports to the class:

```
import cucumber.api.CucumberOptions;
import org.junit.runner.RunWith;
import cucumber.api.junit.Cucumber;
import cucumber.api.SnippetType;
```

   And add Cucumber Options as follows

```
@RunWith(Cucumber.class)
@CucumberOptions(features = "features",
    format = {"pretty",
            "html:target/site/cucumber-pretty",
            "json:target/cucumber.json"},
    snippets = SnippetType.CAMELCASE)
public class GenericTestRunner {
}
```

4. If you Run your app by right clicking on this class "Run GenericTestRunner" you should see the output suggesting which steps need to be implemented in order to test this scenario. These are sample code for step definitions, written in Java, which tell Cucumber how to translate the plain English steps from the feature into application methods. Our next step will be to put these snippets into a Java file where we can start to flesh them out.

```
@Given("^I am connected to the WOF database$")
public void i_am_connected_to_the_WOF_database() throws SQLException {

}

@Given("^The vehicle with plate \"([^\"]*)\" exists$")
public void the_vehicle_with_plate_exists(String arg1) throws Throwable {

}

@When("^I update model of this vehicle to \"([^\"]*)\"$")
public void i_update_model_of_this_vehicle_to(String arg1) throws Throwable {

}

@Then("^Next time I retrive information on the vehicle, the vehicle model should be
    ↪  \"([^\"]*)\"$")
public void next_time_I_retrive_information_on_vehicle_the_vehicle_model_should_be(
    ↪ String arg1) throws Throwable {

}
```

5. Create a new java class in your **test/java/steps** package called VehicleTestSteps. This is where we will implement the scenario steps (Given,When, Then). Copy the snippets (Cucucmber gave you, not those we supplied above) into this class. Add this option to **GenericTestRunner** (glue = "steps"). This option is going to tell Cucumber where our test steps are implemented.

6. Add the following cucumber API and junit imports to VehicleTestSteps class:

```
import cucumber.api.PendingException;
import cucumber.api.java.en.Given;
import cucumber.api.java.en.Then;
import cucumber.api.java.en.When;
import org.junit.Assert;
```

7. To implement the first step ("I am connected to the WOF database") all we need to do is open a connection to the database. In the **VehicleTestSteps.java** edit the first step to look as this:

```
@Given("^I am connected to the WOF database$")
public void i_am_connected_to_the_WOF_database() throws SQLException {

    String url = "jdbc:sqlite:lab3.sqlite";
    Connection connection = DriverManager.getConnection(url);
}
```

8. Run your tests by right clicking on GennericTestRunner. You should see the first test pass (if forming the connection worked ok) but all the other tests should be skipped as we haven't implemented them yet.

9. You will notice that in the next @Given part of the scenario Cucumber has recognised that the method "the_vehicle_with_plate_exists" needs an argument. Cucumber has been clever, and has rightfully recognised that we have a plate value in our scenario, therefore it is assuming that this plate number will be an argument for this method. As this is something that is given all we need to do is record this information for now. Let's change "String arg1" to something meaningful first (e.g."String plate"). The code we need here will be something like this:

```
String vehicle_plate = plate;
```

   As we will need plate in other steps it might be better if we add a class level variable to keep this information, so add private *String vehicle_plate =""*; line to the top of your class and change the method to use this variable only *vehicle_plate = plate;*

10. To implement our next @When step we will need to call some kind of method in one of the implementation classes which will update the vehicle table for us. This is the actual code that we want to test. But as we are using ATDD we do not have this method implemented just yet so we will assume that our implementation of update vehicle method will end up in our SQLiteJDBC class which we have worked on last week. Lets add the following lines to our @When test:

```
SQLiteJDBC sqllitejdbc = new SQLiteJDBC();
sqllitejdbc.updateVehicle(connection, vehicle_plate, newModel);
```

    Note, update arguments in the method signature accordingly

11. As you can see we need connection here again, so it makes sense to give the connection a class level scope. Change your first @Given method to allow this.

12. You will see few errors (red lines) in your test file by now. This is due to the fact that we do not have appropriate method in our SQLiteJDBC. Create a method signature for now and run the tests. They should fail. This is so we can have confidence that when the tests pass we are really done. This is outside-in development (similar to TDD).

13. Let's now implement our last test @Then. All this test will need to do is get a vehicle record with given plate number and check if the model has been updated to "Corolla". To do this we will again need to call an implementation method (which we haven't written yet) which simply retrives vehicle record for the given plate. Remember to give method argument a meaningful name. Write your @Then method as follows:

```
sqllitejdbc = new SQLiteJDBC();
ResultSet set =  sqllitejdbc.getvehicle(connection, vehicle_plate);
Assert.assertEquals(set.getString("model"), model);
```

    Note, that we have needed to use SQLiteJDBC again so this might be a good candidate for a class level variable too.

14. Tests will scream at you with red lines again. Add the getVehicle method signature to SQLiteJDBC class and run the tests again. They should be failing still.

15. Implement updateVehicle and getVehicle methods in SQLiteJDBC class. After each method implementation run the tests and see one by one passing.

1.5 One more scenario

Add Cucumber tests for the following scenario:

```
Scenario: Delete a vehicle
  Given The vehicle with plate "HVJ150" exists
   When I delete the vehicle from the database
   Then All related vehicle inspections should be removed too
```

2 Mock Objects

2.1 Introduction

Mock objects are particularly useful in situations where "every class must be implemented before the others" — making it hard to develop and test a single class in isolation. In such cases, mocking frameworks allows the other classes to be substituted by interfaces, corresponding to the public interfaces the "missing" classes will eventually have. When using mocking objects in unit tests we specify Expectations that describe how the mock objects should (pretend to) behave1. The "real" test code is then written with no knowledge of the mocked environment internal detail — and remains unchanged when/if the mockery is subsequently replaced by real objects.

Mockito, for instance, allows the developer to define the expected behaviour for the methods of a class or interface before they are fully implemented; only method stubs (for a class) or signature (for interface) are required to begin testing. There are a number of other available mocking frameworks, for Java projects, such as jMock (jmock.org), and besides we had chosen to use Mockito for this lab, due to its clean and simple syntax, you are highly recommended to try a couple of others on your own time so that you can pick you preferred one. There's a comparison of jMock and Mockito at *http://zsoltfabok.com/blog/2010/08/jmock-versus-mockito/* and there are lots of similar articles.

2.2 Your tasks for this lab

This task involves using TDD to develop a WofInspection class (and the corresponding WofInspectionTest class). The WofInspection functionality will be closely connected with that of the Test interface — which hasn't been written yet. However, by using Mockito and writing the appropriated method stubs, the specifics of Test can be deferred until later and TDD development of WofInspection can proceed in the meantime.

1. Have a look at below links so that you can improve your knowledge on how Mockito framework works. Explore some of the features Mockito offers — particularly mock() and verify(). Try to understand what an argument matcher is and how it can help you develop effective tests using mock objects.
   - http://site.mockito.org
   - http://static.javadoc.io/org.mockito/mockito-core/2.7.13/org/mockito/Mockito.html#1
2. In your WofInspection project from last week add Mockito dependency in pom file as follows:

```
<dependency>
   <groupId>org.mockito</groupId>
   <artifactId>mockito-core</artifactId>
   <version>2.7.13</version>
   <scope>test</scope>
</dependency>
```

3. Add the two new classes listed below to you project. Please, pay attention to place them to the right folder.

```
./src/main/java/CarWolfInspecption.java
./src/main/java/TestResul.java
./src/test/java/CarWolfInspectionTest.java
```

4. Get familiar with these two new classes, run the tests and update it as necessary so that the test passes.
5. Update base test method in order to complete all inspection' checks:
   - checkTyreCondition() must be called 5 times;
   - checkDoor() must be called at least 3 times;
   - checkBreak(), checkStrucuturalCondition(), checkWindscreenWasherAndWiper(), checkSpeedometer(), checkFuelSystem(), checkSteeringAndSuspension() and must be called once;
   - checkExhaust() must be called at least once;
   - checkLight() must be called between 4 and 6 times;
   - checkGlazing() must be called at least 4 times;

6. Update the checkTyreCondition() method (in the CarWolfInspection class) adding a parameter to receive the tyre's tread depth (an double argument). You will also need to update the test do be consisted with this new context. Note that 'isCompleted()'should pass regardless of the parameter values.
7. Update your test class adding an air bag verification for cars fabricated from 2001 on. In such cases, inspection will be completed whether checkAirBag() is called at least once.
8. Start to think about each of the individual tests, write a new test to verify whether tyre's tread depth is acceptable. According to the NZ Transport Agency, the minimally acceptable tyre's tread depth of 1.5 mm, but may want to consider adding an extra quantity as it should stay higher than this minimal legal limit for the next 6 or 12 months (depending on the wolf duration.) *Hint: you will need to use doubleThat() and an argument matcher for this last challenge*

2.3 Extra lab tasks *(if wanted)*

If you liked to play with Mockito and are keen to practice more, we recommend you to:

1. Write a Test method for each of the inspection's tests (as we did for the tyre's condition);
2. Implement the real CarWofInspection class and replace the mock object for a real inspection object. Your tests should continue passing.

3 Extension

1. Investigate alternative mocking (JMock) and ATDD/BDD (Concordian) tools and try to understand their differences. For instance, does jMock treat expectations in the same way as Mockito? Are there any significant differences between the ways these tools approach the mocking process?

   If keen to go further on this, you could even repeat the suggested exercises from the previous section — or try a suitable example of your own. Do you have a preferred framework? What are the most important reasons for your choice?

   Note that it is recommended that you **not** use more than one mocking framework in a project: they don't always play nicely together and differing mock syntaxes can be confusing.

2. Think about how you could use mocking and ATDD while working on tasks in your other projects e.g. SENG302.