# Part 2 – Web Testing and Model Based Testing

## System chosen

Mention link and issues caused

For this task, the choice was made to use the Agenda textbooks website. This website had the listed functionality provided (logging in, product search, adding and removing products, checking out and logging out) and so was deemed fit to use for testing.

https://ps.agendatextbooks.com/en/login.php

Unfortunately, at a much later stage in the testing process it was noted that the site lacks the functionality to view the details of the product, that is later needed in the system. Time lost addressing this issue led to some of the cases below not being fully implemented, but method stubs were created in order to continue demonstrating the testing principles, as will be outlined in this report.

Note that this task made use of the chrome driver situated within the webtesting folder. This might need to be altered to the specifications of the browser installed on the machine.

The

System.*setProperty*("webdriver.chrome.driver", "C:\\Users\\larab\\OneDrive\\Documents\\chromedriver\_win32\\chromedriver.exe");

Line also needs to be altered to point to the path of the driver on that device.

## Task 1 – Automated tests

Throughout this task the use of the Page Object design Pattern. This involves creating a Page Object (class) was each required webpage, containing all the methods related to the tasks performed on that page. Whenever an interaction with the page is required, methods from the page object are used. This prevents code duplication and makes test design easier to maintain.

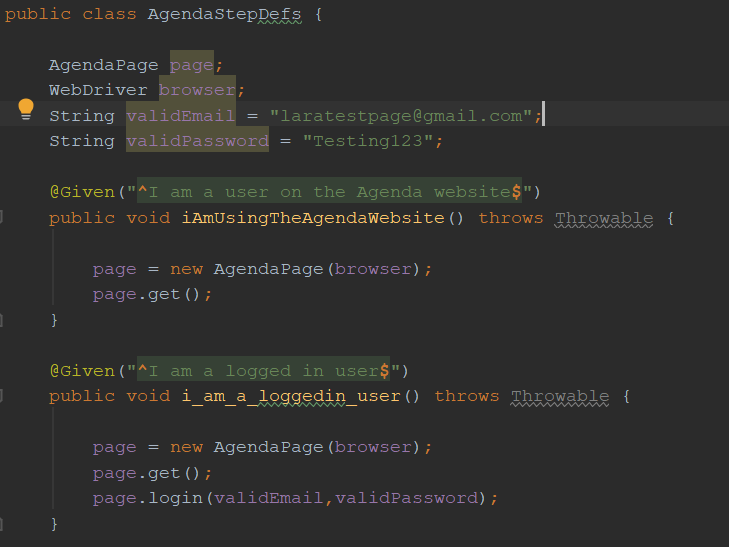
**AgendaPage**

This class is responsible for running the interactions with the interface. The login and checking if the user has been logged have been successfully implemented, but other methods are not fully implemented, many are just code stubs in order to be able to continue demonstrating the testing principles.

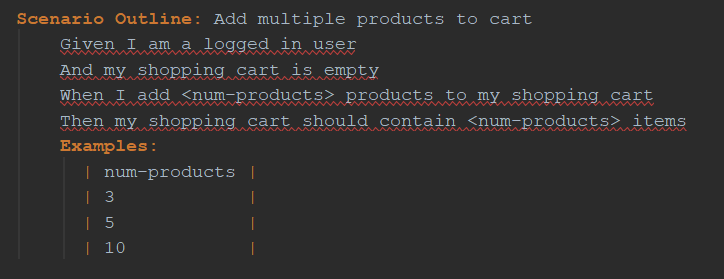
**AgendaStepDef**

Following the Page Object model mentioned, an AgendaPage model is created in the AgendaStepDefs, and made use of to interact with the webpage. Each Step Definition is annotated with either @Given , @When, @Then, following the feature layout. The annotation matches the steps in the feature file.

*Example of stepdefs*



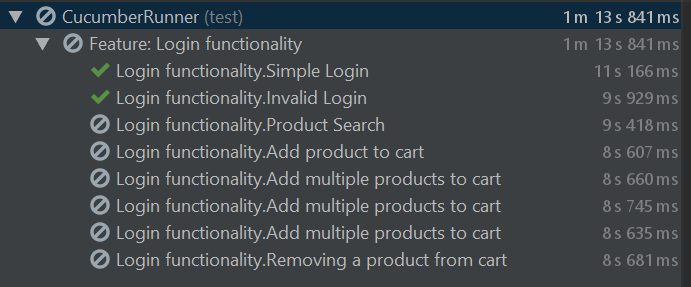
**Agenda.feature**

The feature file contains a set of scenarios implemented from the given task. The steps listed correspond to the methods written in the previously mentioned AgendaStepDef file. Within the 5th Scenario one can note the use of Scenario outlines, that allow for multiple tests of the same format to be run, by making a change to one condition. 

As previously mentioned, only the running of the first 2 scenarios runs successfully, as parts of the other methods were not successfully implemented.

### Running the Cucumber runner

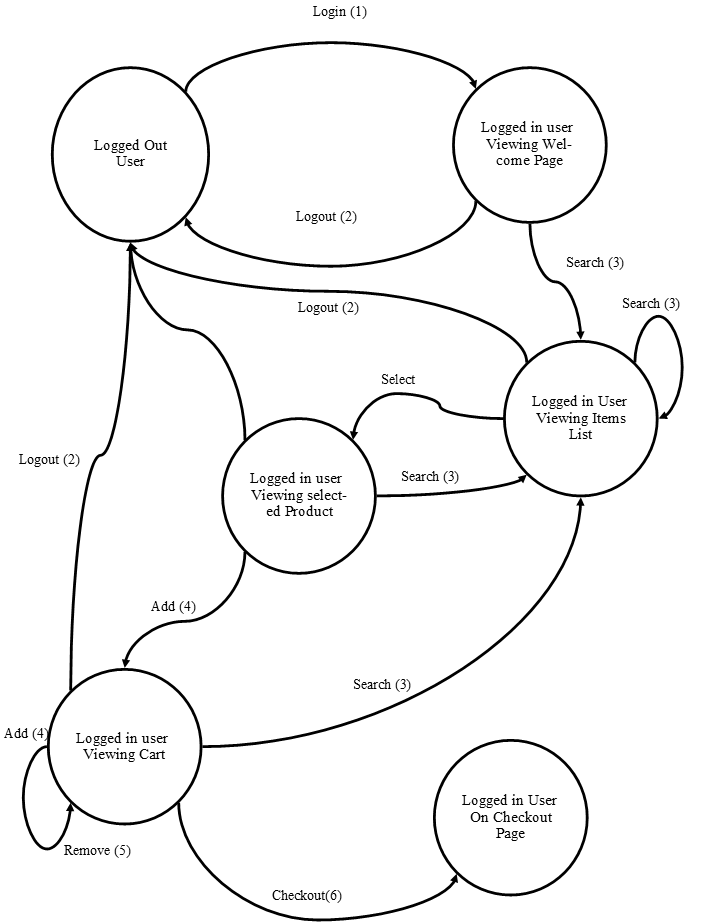
As previously mentioned, the first 2 scenarios were implemented fully, while the rest where represented with method stubs. The functioning scenarios can still be run as part of the suite. Running the cucumber runner gives the following results.



## Task 2 – Model Based testing

### Model Created

The model designed is the following



While creating the model, the following assumptions and design decisions were made:

* A user will not logout after checkout (as it was given in the question that interaction ends after checkout)
* Assumed that searches always result in a valid output
* Assumed that the adding actions takes you to view the cart page automatically

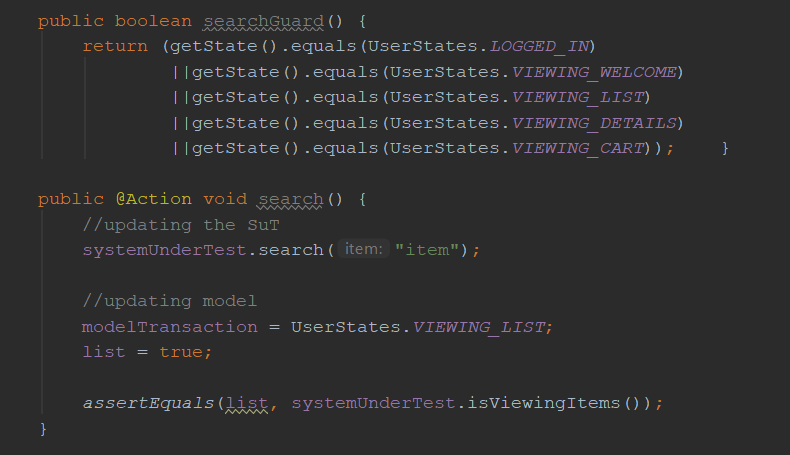
### Converting to ModelJUnit

Despite the previously mentioned issues in implementing some of the AgendaPage functions, the Model tests were still implemented, to demonstrate the testing principles.

The conversion form the graphical model to the test suit was done by first examining the states involved (present on the nodes of the graphical model) and representing them as enums in the UserStates enum.

Within the AgendaUserModelTests, each action (arrow/ branch in the graphical model) was represented by an annotated method. Before each method call a methodGaurd was also implemented, which ensured that the model is in the right state to execute that action. Being in the right state would be equivalent to being in a state that has the option (arrow going out of it) for that particular action to be performed.

An example of this is the following, showing how the search action can be performed from any state except the LOGGED\_OUT and CHECKED out state.



It is noted that when some of the method stubs are called, a null pointer exception is thrown due to the lack of interaction and functionality.

