L-Università ta' Malta
Faculty of Information & Communication Technology

Department of Computer Information Systems

CPS3230 - Assignment

Philip Paul Grima(22602H) *
*B.Sc. (Hons) Software Development, 3rd year

Study-unit: **Fundamentals of Software Testing**
Code: **CPS3230**
Lecturer: **Dr Mark Micallef**

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as “the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I, the undersigned, declare that the assignment submitted is my work, except where acknowledged and referenced.

I understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Philip Paul Grima	
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature

CPS3230	CPS3230 - Assignment
Course Code	Title of work submitted

08/11/2022

Date

Introduction

For task 1 of this assignment, a Web driver with selenium was used to achieve a well-implemented screen scraper. Along with the provided webpage which stores the alerts, Malta Park is the chosen e-commerce website and it is utilized every time a new alert needs to be uploaded. Continuing to task 2, Cucumber was used to implement all the required test suites. For both tasks, the fundamentals portrayed in the online tutorials were followed.

Task 1

Implementation

Unit testing and test-driven development was the main approach taken when implementing the correct functionality and the adjacent test cases to develop a high-quality screen scraper program.

Before any implementation of the functionality, a test case was added and the functionality was developed to fulfill the given case. For example, consider below *Figure 1.* The test case matches the functionality so when an undefined type of product is entered an empty string is declared as the input and the program returns false.

```java
@Test
public void testSearchWithEmptyString() throws IOException, InterruptedException {
    //Setup
    //Deleting all the previous requests in order to make sure that the 5 sent are of this test.
    httpDeleteRequest httpDeleteRequest = new httpDeleteRequest();
    httpDeleteRequest.sendDeleteRequest();
    //Mocking the status of the page.
    StatusProvider statusProvider = Mockito.mock(StatusProvider.class);
    //Mocking the status of the request.
    RequestStatusProvider requestStatusProvider = Mockito.mock(RequestStatusProvider.class);
    //Mocking the status of the driver.
    DriverStatusProvider driverStatusProvider = Mockito.mock(DriverStatusProvider.class);
    //Setting the page mock to successful.
    Mockito.when(statusProvider.getStatusProvider()).thenReturn(StatusProvider.ONLINE);
    //Setting the request mock to successful.
```

Test class

```java
//Creating a number of different strings for each option.
String[] Car = {"Toyota", "Mazda", "Honda", "Suzuki"};
String[] Boat = {"Yacht", "Sail boat", "Fishing boat", "Dingy"};
String[] propertyForRent = {"Apartment to let", "Garage to let", "Office to let", "Flat to let"};
String[] propertyForSale = {"Apartment for sale", "Garage for sale", "Office for sale", "Flat for sale"};
String[] toys = {"Toy gun", "Toy car", "Toy boat", "Toys"};
String[] electronics = {"Laptop", "iPhone", "Samsung", "Computer"};

//Allocating a search string index.
int x = (int) (Math.random() * (4 + 1) + 0);
//For integer out of bound purposes.
if(x == 4) {
    x= x - 1;
}
//Choosing which type of option to search based on the parameter passed from the test.
String searchString = switch (type) {
    case 1 -> Car[x];
    case 2 -> Boat[x];
    case 3 -> propertyForRent[x];
    case 4 -> propertyForSale[x];
    case 5 -> toys[x];
    case 6 -> electronics[x];
    default -> "";
};
```

screenScraper class

***Figure 1**: The above shows the test with its allocated functionality. By developing the test first, it was easier to know what needs to be developed and what the goal is. Not to mention that once everything was done, the function was tested in order to make sure that it is safe to proceed.*

By taking a test-driven approach, the program was built piece by piece making sure that the current functionality present is indeed working by creating the allocated tests beforehand. The development only proceeds once the correct functionality has been implemented for the given tests and the desired conclusions have been reached.

Unit tests were also implemented to ensure that each aspect/case of the program was fully covered. The main aspects covered by the unit tests are; the different types of products that can be searched, the state of the inputted string, and also the states of the website, request and driver (online/offline). In order to achieve this last one, mocks in the form of Mockito were used to facilitate and simulate this interaction. Refer to the section below about test doubles to view this aspect in more detail.

Test patterns were used mainly in the form of dependency injection. In *Figures 1, 4 & 7* it can be seen how all aspects; parameters, constructor, and setter injections were used in order to achieve the desired tests.

Some constant difficulties that were recorded include the loading and visibility of certain on-screen elements. If the required element has not loaded yet and the program tries to find or interact with it, the compiler would throw an exception and the program would fail. This was mainly handled by creating both *WebDriverWait* objects and timeouts. The former was used to wait for elements to become visible or clickable, while the latter allows the current website to load in full which increases its responsiveness. Refer to *Figure 2* below.

```
driver.manage().timeouts().implicitlyWait( t 100, TimeUnit.SECONDS);
WebDriverWait wait = new WebDriverWait(driver, timeOutInSeconds: 100);
wait.until(ExpectedConditions.presenceOfElementLocated(By.id("cookiebar")));
```

***Figure 2**: Both timeouts and web driver objects are used to make sure that the desired element is indeed present, preventing the program from throwing an error.*

Furthermore, it is important to note that the status of both the POST and DELETE requests along with the body was displayed in the console each time in order to get a better understanding of what is going on. Refer to the below *Figure 3*.

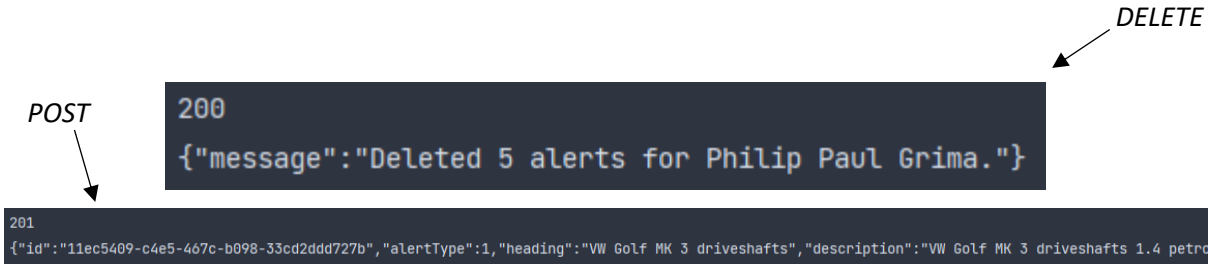

***Figure 3**: The above shows the message displayed by the console after a successful request. The former portrays a DELETE request while the latter is a POST.*

Test doubles

In this assignment, the test doubles used were mocks using Mockito. This framework allows you to simulate an entire instance with only a few lines of code. In this case, Mockito was used to simulate when the website, request, and driver were online and offline since this is a very probable possibility in any online site. *Figure 4* illustrates how the situations were split into instances and Mockito was used to simulate each instance. Depending on what the mock was set to simulate, the program would either return a true or a false.

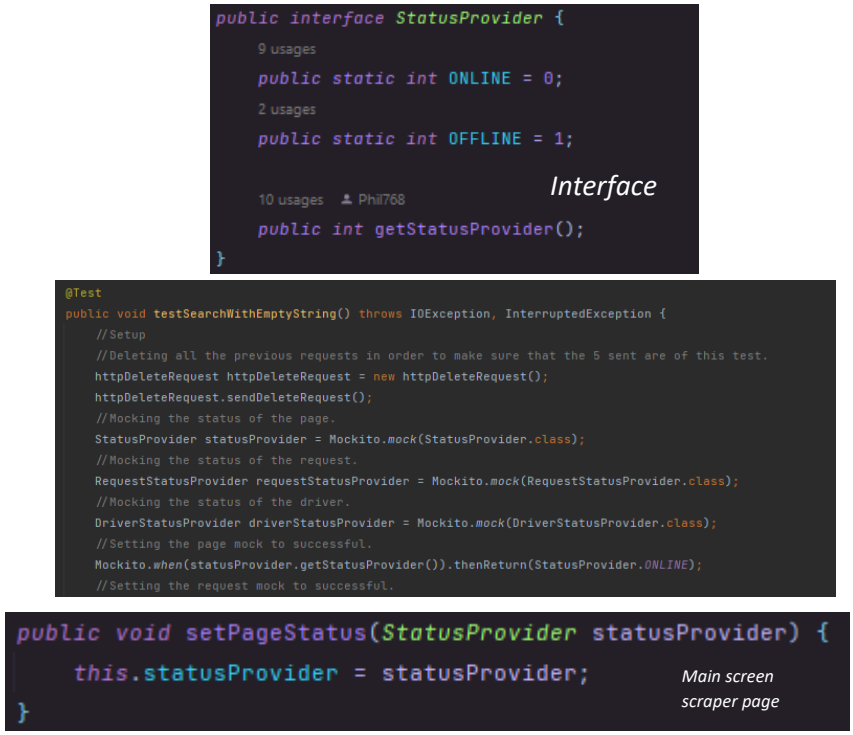

***Figure 4**: The above shows how easy it is to simulate an extreme condition by using Mockito.*

Analysis

The screen scraper has a total of 13 tests which cover testing for each option, inputting an incorrect string, and also a test for each mock when the desired outcome is a failure or null. The tests all work as expected.

In terms of code coverage, the screen scraper manages to cover all the instances of the class which resulted in 100% code coverage. This was done by making sure that all the parts of the code were covered by creating the required tests. It has already been mentioned that 13 tests were created which covered all the instances which could occur, even when mocked objects appear to be null, imitating special situations. *Figure 5* below gives a good description of the statistics of the class are;

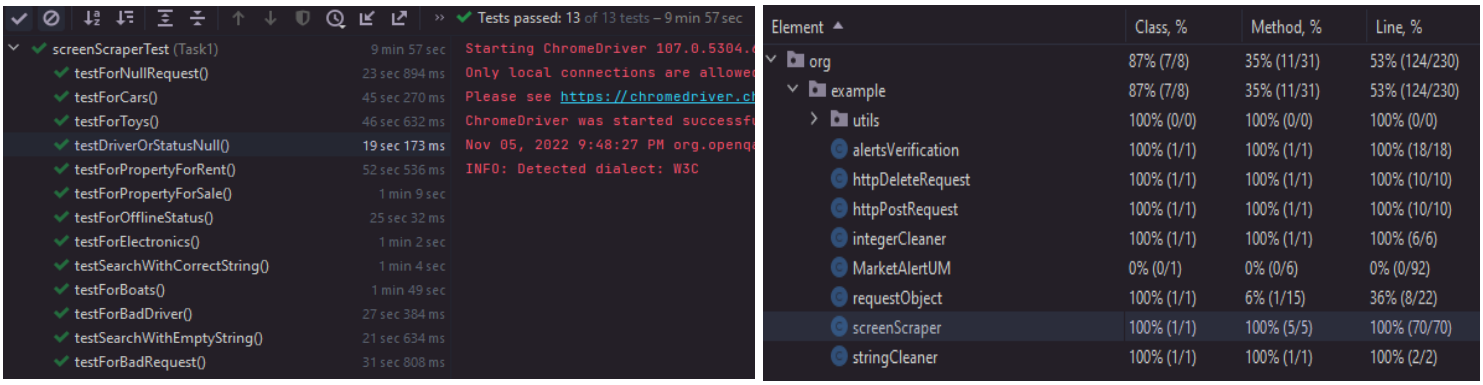

Figure 5: *All 13 tests pass and screenScraper, the class which handles all the tests, gives 100% code coverage by including all the lines of code of the scraper in the allocated tests. It is important to note as well that all the other classes which are utilized by the class also give a 100% coverage*

It must also be noted that Malta Park recently added an alert that requires the user to wait for 10 seconds before it is able to be clicked away. Despite the fact that using threads is not always a best practice, in this case, it was the most optimal way since it forced the scraper to wait for 11 seconds (1 extra second to ensure visibility of elements) before it interacted with the alert. Since this needs to happen before every test, it is present in the test class (*In task 2 it is very similar).*

```
//Manually putting the system on hold for 11 seconds
Thread.sleep( millis: 11000);
```

Figure 6*: Manually putting the scraper to sleep in order to wait for the alert timer to finish and allow the required button to become visible.*

Task 2

Implementation

For this task, the Cucumber framework with web driver was used in order to achieve the requested test suites concerning online screen scraping. Once again, a test-driven approach was taken by making sure that the tests in both the feature and the steps files were developed prior to the methods which contain the main functionality. When there was a scenario that required access to the alerts on *MarketAlertUM*, it was assumed that the input ID should be part of the Cucumber *When* tested as a String and later on passed as a parameter to the method. This same method was used whenever there was a need to access the alerts.

In this task, constructor injection was the main test pattern used. In below *Figure 7*, it can be seen how the Web driver is passed as the only parameter to the given class, depending on which website it needs to access. The other injections were used as well however this is the most prominent one

Figure 7: *The Web driver is first initiated at the beginning of the cucumber steps class and later on allocated according to the object passed in through the constructor in the main class.*

Testability of the website

The main problem with the testability of the website is the fact that most of the elements which make a standard alert are either not in a specific tag, or are not part of a particular class. This was really an issue through task 2 since it mainly involved testing the overall website, rather than interacting with the e-commerce site. In *Figure 8* below one can clearly see how both the price and the description are not even contained in any tag, they just make part of a table of data. Despite the fact that this method still functions, it makes it much harder to target specifically using a web driver. One can also see how most of the tags do not have classes or unique IDs. This made it quite hard to target and only left XPath as a method to obtain, which is not always reliable. The best recommendation to give is to add specified classes for each section of the alert to be able to target better, or even better unique IDs.

In a particular scenario, it was required to develop a test case that confirmed that all the elements which make a common alert are present. Given the below Figure, it was quite hard to accomplish and the chosen method, especially for the price and description which do not form part of any tag, is unusual.

```
▼ <tr>
  ▼ <td colspan="2">
    ▼ <h4>
        <img src="/images/icon-car.png" width="100">
        " Toyota Celica 2000 exhaust "
        <font color="red">(3 minutes ago)</font>
      </h4>
    </td>
  </tr>
▼ <tr>
  ▼ <td rowspan="4" valign="middle" align="center">
      <img src="https://www.maltapark.com/asset/itemphotos/9514624/9514624_1.jpg?_ts=1" width="200">
    </td>
  </tr>
▼ <tr>
    <td>Toyota Celica 140BHP - EXHAUST PIPE</td>
  </tr>
▼ <tr>
  ▼ <td>
      <b>Price: </b>
      " €6000.00"
    </td>
  </tr>
▼ <tr>
  ▼ <td>
      <a href="https://www.maltapark.com/item/details/9514624">Visit item</a>
    </td>
  </tr>
```

***Figure 8**: None of the elements form part of a class or have a unique ID. This makes them much harder to locate since the main option left is to use their absolute path with XPath.*

Link to Google Drive containing the Cucumber recording:

https://drive.google.com/drive/folders/1Pd8Ic6vM0Akn3pyk4guIWgvLOjcxMLt7?usp=sharing

Link to GitHub repository containing all the code of this assignment:

https://github.com/Phil768/University/tree/main/School/SoftwareTesting/CPS3230%20-%20Assignment1