

# CPS 3230

## Fundamentals of Software Testing

### Assignment documentation

Lara Brockdorff  
110899M

## Table of Contents

Part 1 – Unit testing .....	3
Task 1 - Setup .....	3
Task 2 – Initial unit testing .....	3
Testing the system undertest .....	3
Commenting on the coverage.....	3
Noted testability Issues.....	5
Task 3 – Advanced unit Testing.....	6
Part 2 – Web Testing and Model Based Testing .....	8
System chosen .....	8
Task 1 – Automated tests .....	8
Running the Cucumber runner .....	9
Task 2 – Model Based testing .....	11
Model Created .....	11
Converting to ModelJUnit.....	12
Part 3 – Theoretical Foundations.....	14
1. Control Flow Graph.....	14
2. Test Suite with 100 % Statement Coverage.....	15
3. Test Suite with 100 % Statement and Branch Coverage.....	15
4. Paths for 100% boundary Interior path coverage.....	16
5. Data Flow Graph .....	16
6. Data Flow adequacy.....	17
7. Static Code analysis.....	17
8. Issues detected .....	18
Conclusion.....	18

Please note that all related code can be found in the attached zip copy of the git repository, as well as at:

[https://github.com/LaraBrockdorff/CPS3230\\_SoftwareTesting](https://github.com/LaraBrockdorff/CPS3230_SoftwareTesting)

## Part 1 – Unit testing

### Task 1 - Setup

As required, a Git repository was created on github, and the user **mark-uom** was added as a contributor in order to have access. The repository can be found at :

[https://github.com/LaraBrockdorff/CPS3230\\_SoftwareTesting](https://github.com/LaraBrockdorff/CPS3230_SoftwareTesting)

Commits were made to this repository when changes were made.

In order to easily identify the differences between tasks 2 and 3 of this past, a release was set after the end of task 2.

In order to get familiar with the system it was first run from the runner present, and tested manually. Through this process, it was noted how this was not possible to test all the possible edge cases in a structured way. Understanding how cumbersome and easily fallible it would be to go through all this manual process every time, shed light on how value writing structured tests is.

### Task 2 – Initial unit testing

#### Testing the system undertest

While writing unit tests for the system under test, the principles behind unit testing were kept in mind, and tests would only test one specific thing, and will always return the same result, independent of other tests. This approach of independent tests also makes debugging future issues that arise when changes are made, and only the concerned points will fail (if bugs are present) and not other dependencies. These tests were kept simple and well commented to even act as documentation.

#### Commenting on the coverage

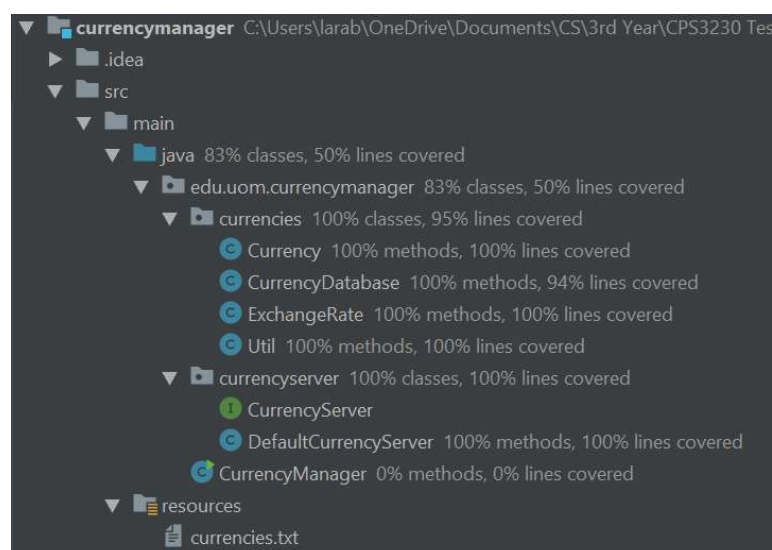


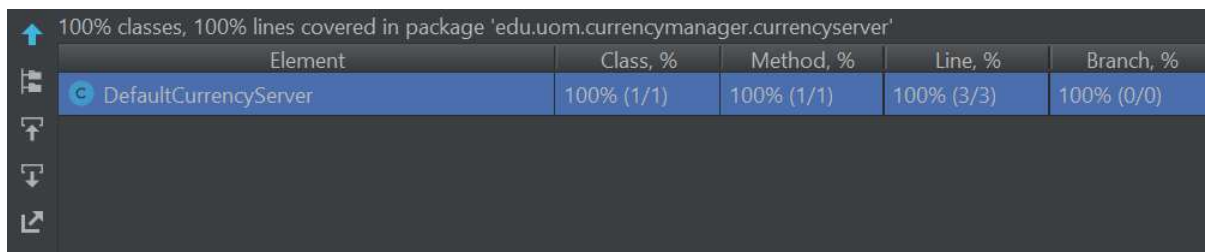
Figure 1 Initial coverage results

## Class coverage

The first thing which was noted is that a class coverage of 83 % was initially achieved. This was due to the fact that the *CurrencyManager* runner class was not tested. This is because the runner in the main menu is dependant on user input with various possible paths, and the other helper methods are dependant on interactions with the *CurrencyDatabase* class. Extracting more of the logic from the methods in this runner class would increase overall coverage.

## CurrencyServer package

This package consists of the *CurrencyServer* interface as well as an implementation called *DefaultCurrencyServer* calls that makes use of a random number generator inside the class within the *getExchangeRate* method, making the returned value untestable. Despite this, a test that merely invokes the method and asserts that the value returned is not null was implemented to assert that any errors that could cause the program to crash while that method is being executed will be caught.



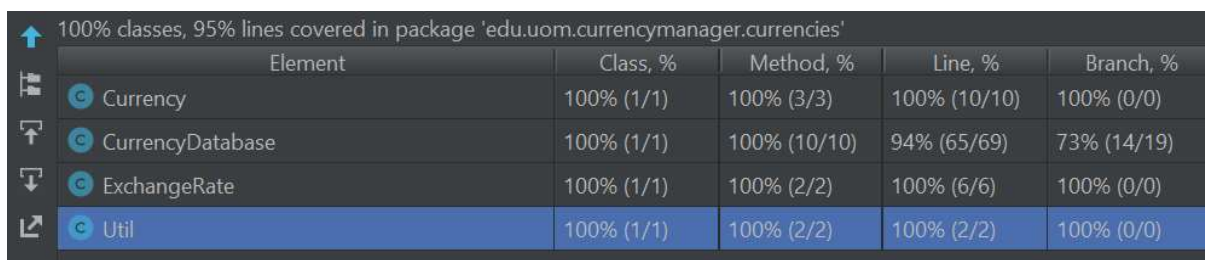
100% classes, 100% lines covered in package 'edu.uom.currencymanager.currencyserver'				
Element	Class, %	Method, %	Line, %	Branch, %
DefaultCurrencyServer	100% (1/1)	100% (1/1)	100% (3/3)	100% (0/0)

Figure 2 Server Initial Coverage

The fact that this class has full coverage, but yet its internal workings are not properly testable, sheds light on how coverage results are in no way a guarantee that the system is bug-free.

A test dummy of this method was created where the random value was replaced with a fixed value, in order to be able to predict the returned value with confidence in order to test the methods that make use of the class.

## Currencies package



100% classes, 95% lines covered in package 'edu.uom.currencymanager.currencies'				
Element	Class, %	Method, %	Line, %	Branch, %
Currency	100% (1/1)	100% (3/3)	100% (10/10)	100% (0/0)
CurrencyDatabase	100% (1/1)	100% (10/10)	94% (65/69)	73% (14/19)
ExchangeRate	100% (1/1)	100% (2/2)	100% (6/6)	100% (0/0)
Util	100% (1/1)	100% (2/2)	100% (2/2)	100% (0/0)

Figure 3 Currencies Package initial Coverage

## Util class

This util class containing only the *formatAmount* method was tested by testing the outputted formatting of different numbers (23.567, 3.4, 0000) which required rounding, adding of a last decimal place, and formatting of excess 0 values respectively. These numbers were chosen from different

regions of the number set (different decimal place formats) valid in this context since it is not feasibly possible to test all possible doubles.

This class has full method, line and branch coverage.

#### ExchangeRate class

In this class, only the *toString* method was tested, even though not much logic was present in it. This was done by passing the test data and asserting that it matches the output of the *toString* method.

This class has full method, line and branch coverage

#### Currency class

This class is mainly used to create the object type of currency which contains a *fromString* method and a *toString* method. These were both tested by populating the objects with test data and comparing the output. In the case of the *fromString* method a valid currency was first tested to make sure it was populated correctly, then an invalid currency containing null data was passed to assure that a null pointer exception is thrown as expected.

This class has full method, line and branch coverage.

#### CurrencyDatabase class

Similarly to the previous cases, the contained methods were tested by asserting that returned results matched the expected results. This class posed challenges while being tested due to the way the methods interact with CurrencyServer and with the currencies.txt (database).

Due to the interaction with the currencies.txt file, the init method was not directly tested, reducing both the line and branch coverage in the class since not all the possibly thrown exceptions were tested for, as testing for this would involve manually altering the text file, which would go against the structure of unit testing.

Within the getExchangeRate method, one branch relating to the timeLastChecked was not tested, reducing the branch coverage. This is because of the way that the time is used in the ExchangeRate class.

Despite principles being followed related to the atomicity of unit tests, the tests do not function if the currency database/txt file is not present in the correct directory. This is due to the design of the system.

This class has 100% method coverage, 94% line coverage and 73% branch coverage.

#### Noted testability Issues

As mentioned above, testability issues include:

- If the currencies.txt file is not present, all the tests in the CurrencyDatabaseTests class fail, making the test dependant on this file.

- The functionality within the `DefaultServer` could not be tested due to the dependency on the Random number generated within the method.
- The `timeLastChecked` value within the `ExchangeRate` class cannot be manipulated easily when tested due to the way it is set from the System time within the constructor method.

### Task 3 – Advanced unit Testing

Aim of this section was to implement changes to the system under test to make the system simpler to test.

In order to make the system more efficiently to test, the following three main changes were effected;

#### 1. Refactor of Random number in *DefaultCurrencyServer*

While originally testing this class, it was noted that its testability is limited due to the use of a random number generator within the `getExchangeRate` method. It was an instance of a hardcoded dependency.

In order to address this issue and **the Dependency Injection-Parameter Injection** design pattern was implemented. This allowed for depended-on-components to be set at runtime from outside the method itself and to be passed as a parameter.

#### 2. Refactor of `ExchangeRateTime`

Similarly to the previous case, the use of `System.currentTimeMillis` in the `timeLastChecked` field was refactored using and **the Dependency Injection of Constructor injection** design pattern, allowing for the `timeLastChecked` value to be passed in the constructor.

This allows for the value to be set from the constructor, independent of the System method.

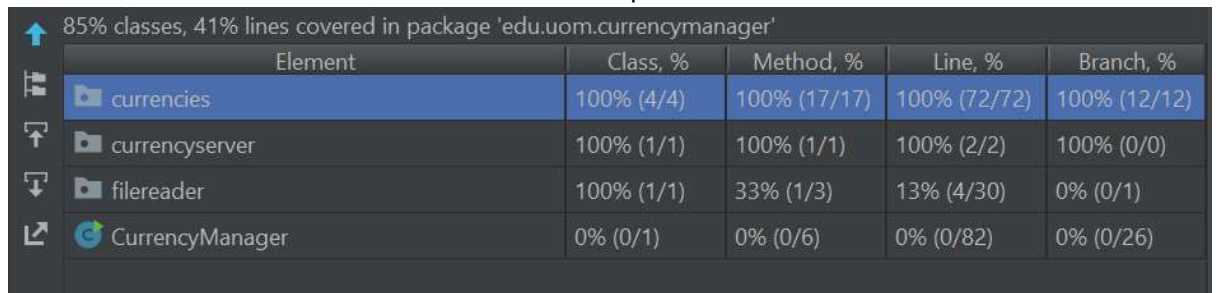
#### 3. Refactor of reading from file

Another issue previously mentioned was that there existed a dependency between the `CurrencyDatabase` class and the `currencies.txt` file being read from. This meant that if this file was not present, none of the related `CurrencyDatabase` would run successfully. This was amended by abstracting the reading of the file to another class, which would later be mocked, using the **DependencyLookup** design pattern.

These changes lead to a large increase in the coverage rates of all classes in the `currencies` and `currencySever` packages, which now have coverage of 100 % (both branch and line)

Despite this, due to the abstraction of the `fileReader` (that was previously being called as a side effect of the `init` method in the `CurrencyDatabase` class), the overall line coverage decreased. Since the file reader class deals mainly with external communication (to an external file), more focus should be given to the integration testing of this component.

Overall, the changes made will also make future testing of the system easier as these changes allow for more modular code and less hardcoded dependencies.



The screenshot shows the IntelliJ IDEA coverage tool interface. At the top, a summary bar indicates '85% classes, 41% lines covered in package 'edu.uom.currencymanager''. Below this is a table with five columns: 'Element', 'Class, %', 'Method, %', 'Line, %', and 'Branch, %'. The table lists four elements: 'currencies', 'currencyserver', 'filereader', and 'CurrencyManager'. The 'currencies' element is highlighted in blue and shows 100% coverage across all metrics. 'currencyserver' also shows 100% coverage. 'filereader' shows 100% class coverage but lower method and line coverage. 'CurrencyManager' shows 0% coverage across all metrics.

85% classes, 41% lines covered in package 'edu.uom.currencymanager'				
Element	Class, %	Method, %	Line, %	Branch, %
currencies	100% (4/4)	100% (17/17)	100% (72/72)	100% (12/12)
currencyserver	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
filereader	100% (1/1)	33% (1/3)	13% (4/30)	0% (0/1)
CurrencyManager	0% (0/1)	0% (0/6)	0% (0/82)	0% (0/26)

Figure 4Updated Coverage results

## Part 2 – Web Testing and Model Based Testing

Please note that all related code and tests can be found both in the Git repo mentioned above, in the A2 folder, as well as in the Zipped Copy of the repo

### 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.

<a href="https://ps.agendatextbooks.com/en/login.php">https://ps.agendatextbooks.com/en/login.php</a>
---

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\\Iarab\\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) for 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*



```

@Given("^I am a user on the Agenda website$")
public void iAmUsingTheAgendaWebsite() throws Throwable {

    page = new AgendaPage(browser);
    page.get();
}

@Given("^I am a logged in user$")
public void i_am_a_loggedin_user() throws Throwable {

    page = new AgendaPage(browser);
    page.get();
    page.login(validEmail,validPassword);
}

```

## 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 5<sup>th</sup> 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.

```

Scenario Outline: Add multiple products to cart
    Given I am a logged in user
    And my shopping cart is empty
    When I add <num-products> products to my shopping cart
    Then my shopping cart should contain <num-products> items
    Examples:
    | num-products |
    | 3             |
    | 5             |
    | 10            |

```

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

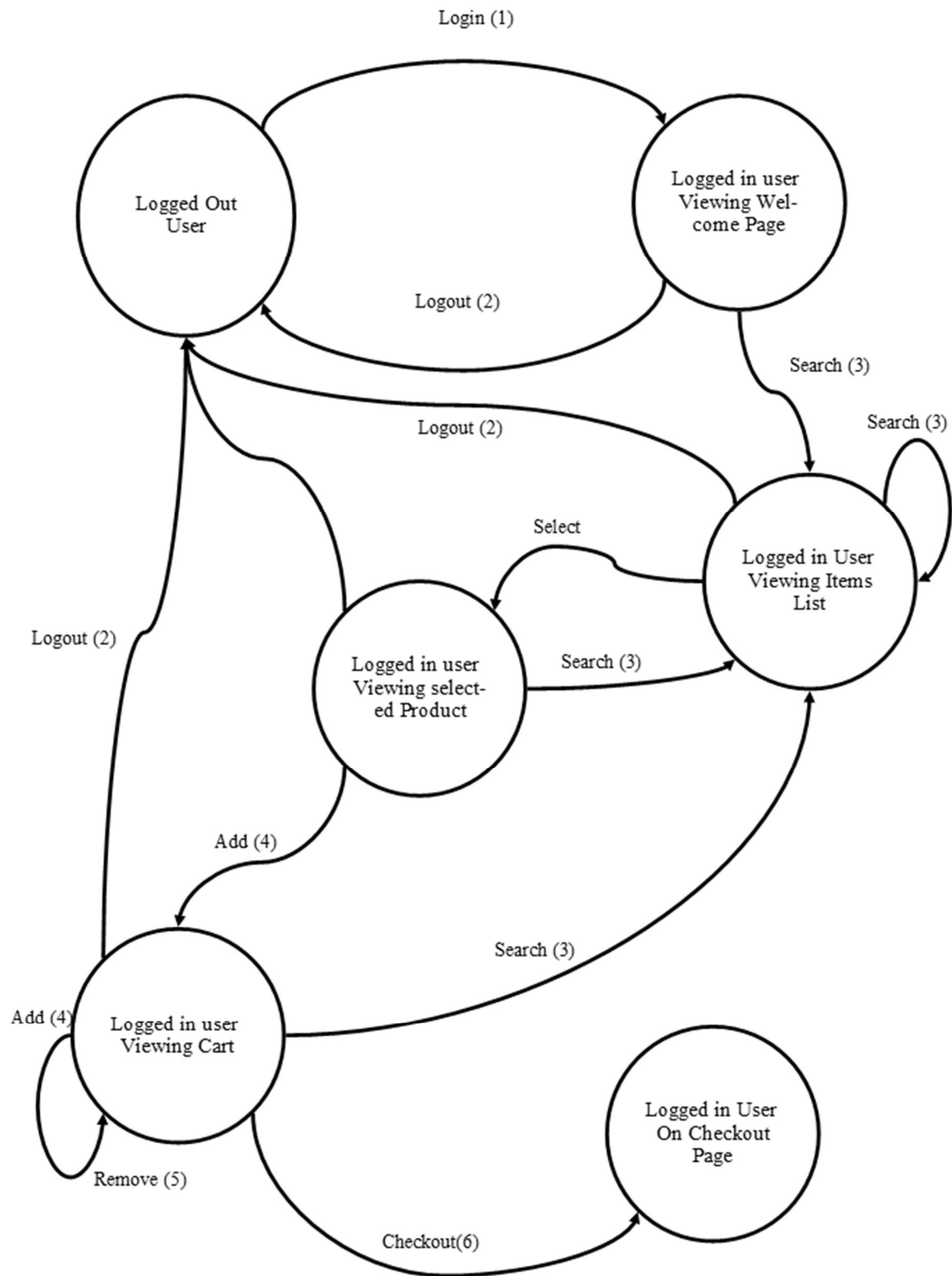
As previously mentioned, the first 2 scenarios were implemented fully, while the rest where represented with method stubs. The functioning scenarios can still be run as part of the suite. Running the cucumber runner gives the following results.

▼	⊗ CucumberRunner (test)	1 m 13 s 841 ms
▼	⊗ Feature: Login functionality	1 m 13 s 841 ms
	✓ Login functionality.Simple Login	11 s 166 ms
	✓ Login functionality.Invalid Login	9 s 929 ms
	⊗ Login functionality.Product Search	9 s 418 ms
	⊗ Login functionality.Add product to cart	8 s 607 ms
	⊗ Login functionality.Add multiple products to cart	8 s 660 ms
	⊗ Login functionality.Add multiple products to cart	8 s 745 ms
	⊗ Login functionality.Add multiple products to cart	8 s 635 ms
	⊗ Login functionality.Removing a product from cart	8 s 681 ms

## 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 from 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.

```
public boolean searchGuard() {
    return (getState().equals(UserStates.LOGGED_IN)
        || getState().equals(UserStates.VIEWING_WELCOME)
        || getState().equals(UserStates.VIEWING_LIST)
        || getState().equals(UserStates.VIEWING_DETAILS)
        || getState().equals(UserStates.VIEWING_CART));
}

public @Action void search() {
    //updating the SuT
    systemUnderTest.search( item: "item");

    //updating model
    modelTransaction = UserStates.VIEWING_LIST;
    list = true;

    assertEquals(list, systemUnderTest.isViewingItems());
}
```

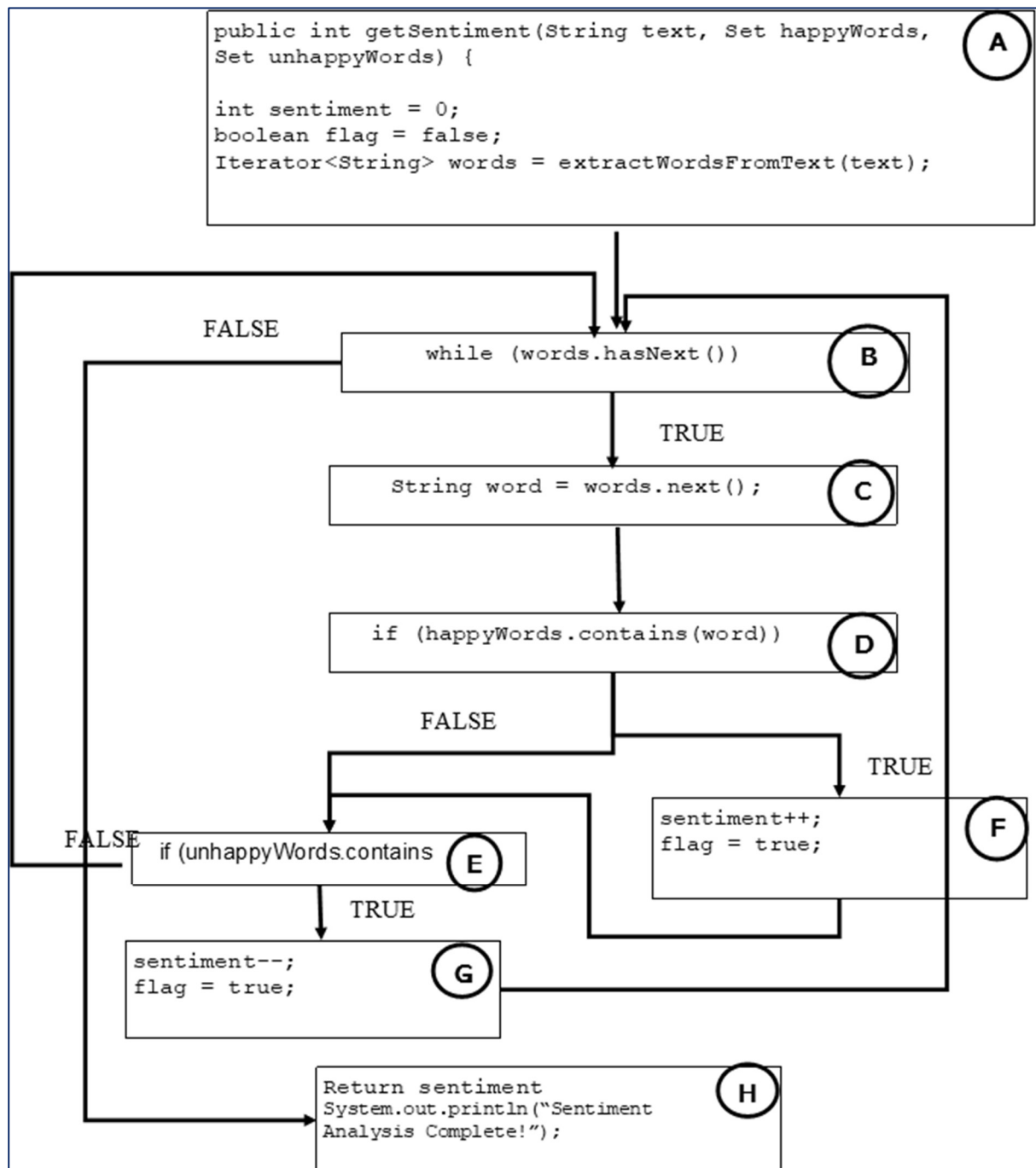
It is noted that when some of the method stubs are called, a null pointer exception is thrown due to the lack of interaction and functionality.

```
done (LOGGED_OUT, removeTransaction, VIEWING_CART)
done (VIEWING_CART, removeTransaction, VIEWING_CART)
done (VIEWING_CART, checkout, CHECKEDOUT)
done (CHECKEDOUT, removeTransaction, VIEWING_CART)
done Random reset(true)
FAILURE: failure in action login from state LOGGED_OUT due to java.lang.NullPointerException
```

## Part 3 – Theoretical Foundations

### 1. Control Flow Graph

A Control Flow Graph (CFG) is a directed graph that models a single procedure. Within a CFG, nodes represent regions of the code, while the directed edges represent possible branches of execution. The given code snippet was overserved and the following CFG was created.



## 2. Test Suite with 100 % Statement Coverage.

Statement coverage is define as :

$$C_{\text{statement}} = \frac{\text{Number of executed statements}}{\text{Number of Statements}}$$

Therefore to achieve 100 % statement overage, all the nodes need to be visited at least once.

This can be done by running a test containing the flowing passed parameters:

Parameter	Content
text	"test"
Set happy	["test"]
Set unhappy	["test"]

Within one loop of execution, each node in the CFG will be executed, but not each branch will be executed, and hence branch coverage will not be 100%. Note that in the nodes D and E, the false branches will not be tested with this case.

## 3. Test Suite with 100 % Statement and Branch Coverage.

This can be achieved with the following test cases

Test One:

Parameter	Content
text	"test"
Set happy	["test"]
Set unhappy	["test"]

Test Two:

Parameter	Content
text	"test"
Set happy	["NOTTest"]
Set unhappy	["NOTTest"]

With the first test case, as mentioned before, all the branches will be visited except the false branches of the D and E nodes. These branches on the other hand and now visited in test case 2, and all branches are covered, achieving 100 % branch coverage.

Another possible testing suit can include one test, with the sets of Happy and Unhappy stated above, joined into one each as follows.

Alternative test

Parameter	Content
text	"test"

Set happy	["NOTTest", "test"]
Set unhappy	["NOTTest", "test"]

#### 4. Paths for 100% boundary Interior path coverage

Since the number of possible paths can be unbounded, different testing criteria can be set to limit the number of paths tested. To achieve 100 % boundary interior path coverage, each path differing by a subpath will be visited. Note that the number of subpaths can still grow exponentially.

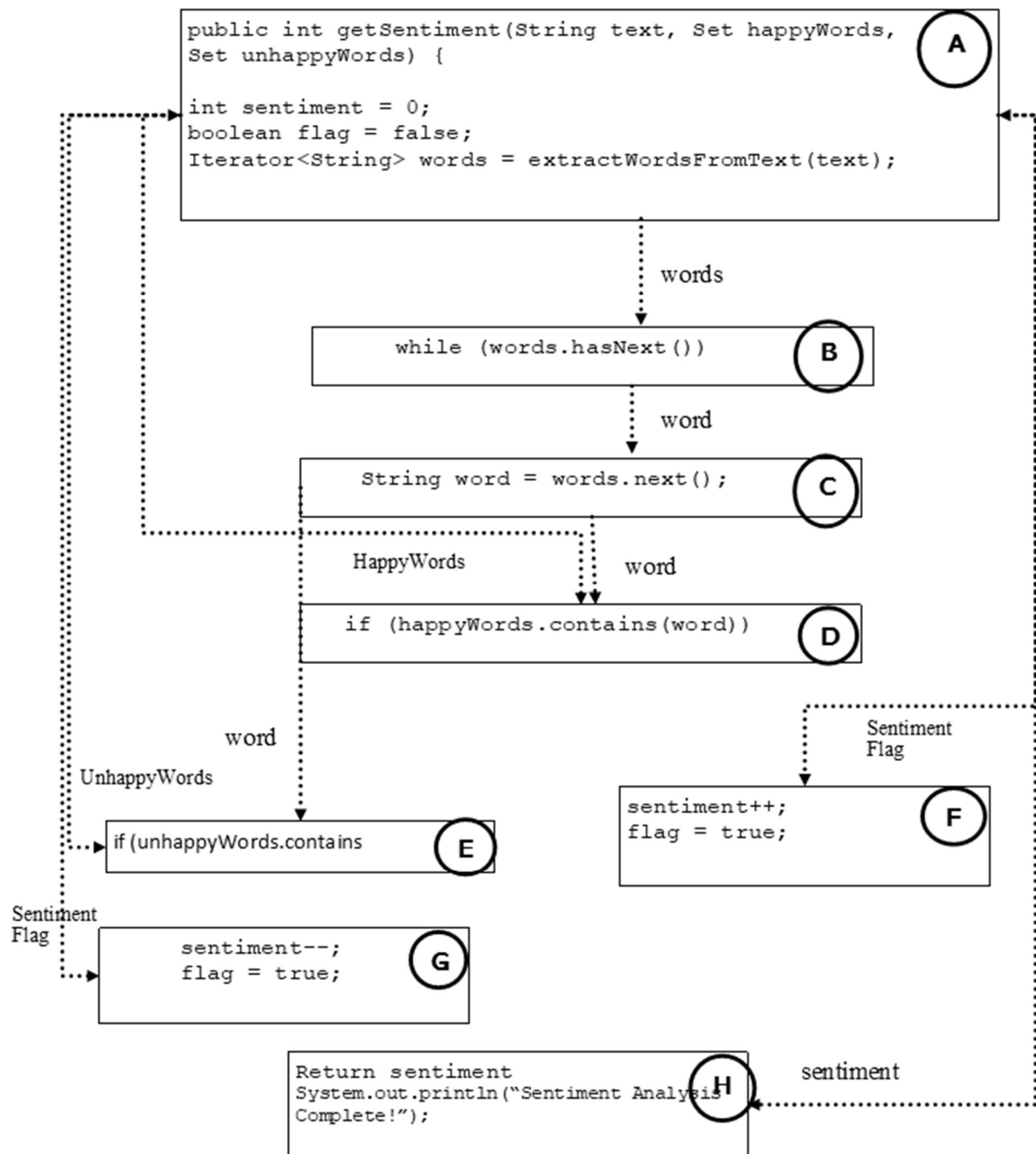
Paths to achieve 100% boundary interior coverage are:

- A -> B -> H
- A -> B -> C -> D -> E -> B -> H
- A -> B -> C -> D -> E -> G -> B -> H
- A -> B -> C -> D -> F -> E -> G -> B -> H
- A -> B -> C -> D -> F -> E -> B -> H

#### 5. Data Flow Graph

A Data Flow Graph (DFG) links the relationships between def-use-pairs. These are pairs of points where data is defined and used respectively.





## 6. Data Flow adequacy

The minimal test suit in Q2 does not full fill the definition clear path criteria since there is no definition clear path since the flag will be reassigned between the nodes F and G.

## 7. Static Code analysis

Static code analysis is a type of analysis that can be performed without running the system, and aims to find faults, ie incorrect step, process or data definition. This can be done in the early stages of the system design, preventing potential bugs to develop further down the line

Static Code analysis be used to compliment/ support dynamic code analysis. This is often required as static testing offers no insight to the intent of the program.

## 8. Issues detected

After reviewing possible manual static code methods such as Requirements Testing, Buddy Checking / Peer Review, Code walk-throughs, it was noted that given the task at hand, not all were possible. For instance the no formal requirements were provided so making Requirement testing impossible. The Buddy checking approach was also not adopted since the work assigned was required on an individual bases. For this reason, a walk though of both the CFG and the DFG was manually carried out, and potential faults were identified. These included:

- The last print statement will never be executed since it occurs after the return statement.
- The **flag** Boolean is never made use of, but is assigned.
- The **word** string is redefined each iteration, instead of be just reassigned.

## Conclusion

Overall this assignment and following report highlights some of the key Unit testing, Web automation Testing, and Model Based testing, as well as examples of CFG and DFG for a given snippet, with evaluation of certain criteria.