# **CSIT5100 Assignment1 Report**

## **1 Introduction**

The reports states the testing information of a Point of Sale (POS) system. Firstly, the report will briefly introduce the business buy-in and refund change system. Then, the report will focus on representing the test result and the analysis.

### **1.1 The Point of Sale(POS) program**

The Point of Sale(POS) program is an business buy-in and refund change system for specific people (i.e., shoppers). The basic functionality of a POS system is helping customers make payments to merchants in exchange of goods or services. Typical POS systems may also offer other functionalities like product management, sales statistics analysis. POS can either read input from your keyboard input or from a specified text file.

### **1.2 Basic Structure of POS**

The Point of Sale(POS) program consists of only one main page -- POS.java. And now let me introduce it.

In the main program, Firstly, merchants can set their username and password, mobile currency(HK or US) and whether to pay taxes via userPasswordFile.txt, and holiday discounts, member discounts and the names, product IDs and prices of various products via productListFile.txt. After entering the POS program, merchants need to enter their username and password to activate the system. After entering the correct user name and password, the system can determine whether the customer has a membership and the register the buy-in products. After registering all the products purchased by the customer, the system will automatically calculate the total price and the amount to be refunded based on the fees paid by the customer. Finally, you can exit the system by entering 2 in the console

It’s more clear to show the structure using a flow chart:

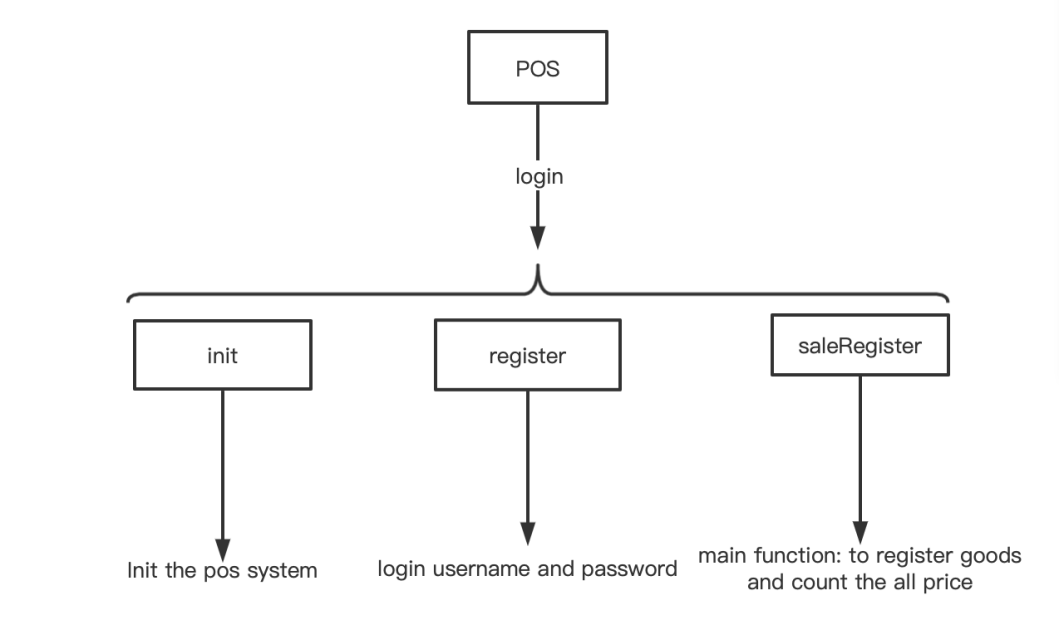


Fig 1. POS Flow Chart

## **2 Coverage**

Overall coverage:

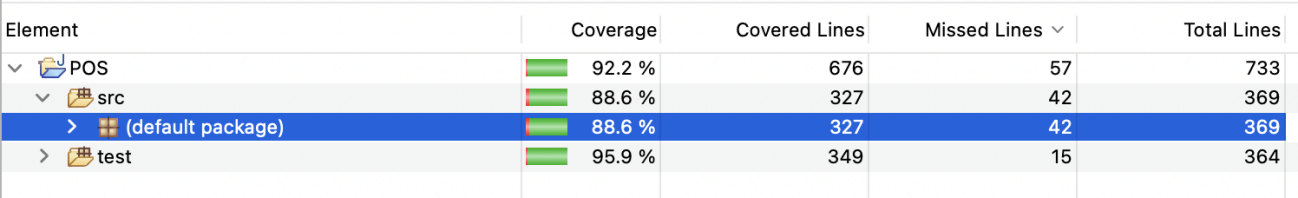


Fig 2. Line Coverage

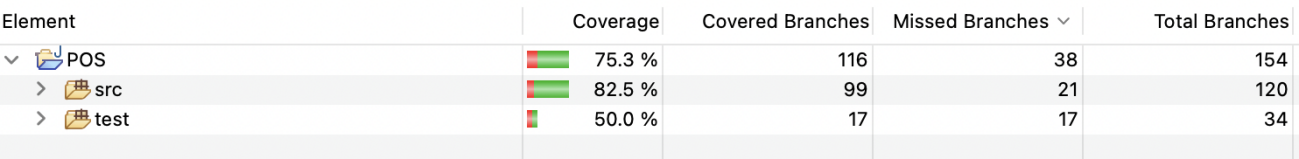


Fig 3. Branch Coverage

The line coverage of the source folder is 88.6%, and the branch coverage is 82.5%.

And line coverage and branch coverage for all the files in source code folder:

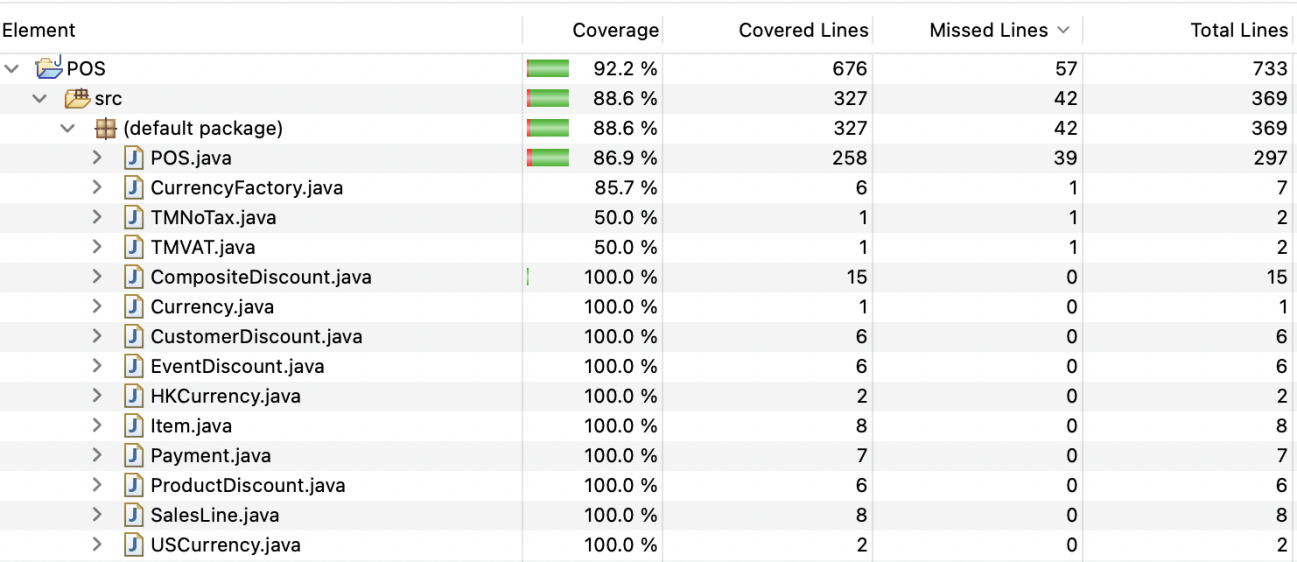


Fig 4. Detailed Line Coverage

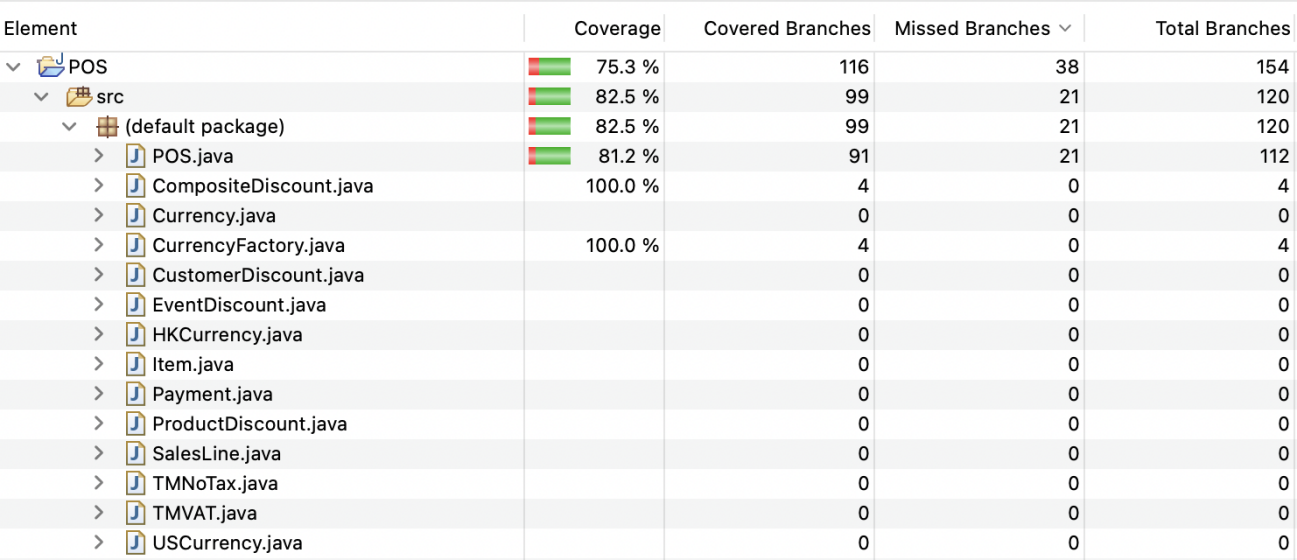


Fig 5. Detailed Branch Coverage

## **3 Constructing the test cases to achieve high coverage.**

Actually I have done a lot repeat work because of lacking of experience in testing. I didn’t focus on TestMain.java(integration testing) at start. Instead I spending a lot of time writing test case for each source file. The following diagram represents the specific function of each java file in the test folder.

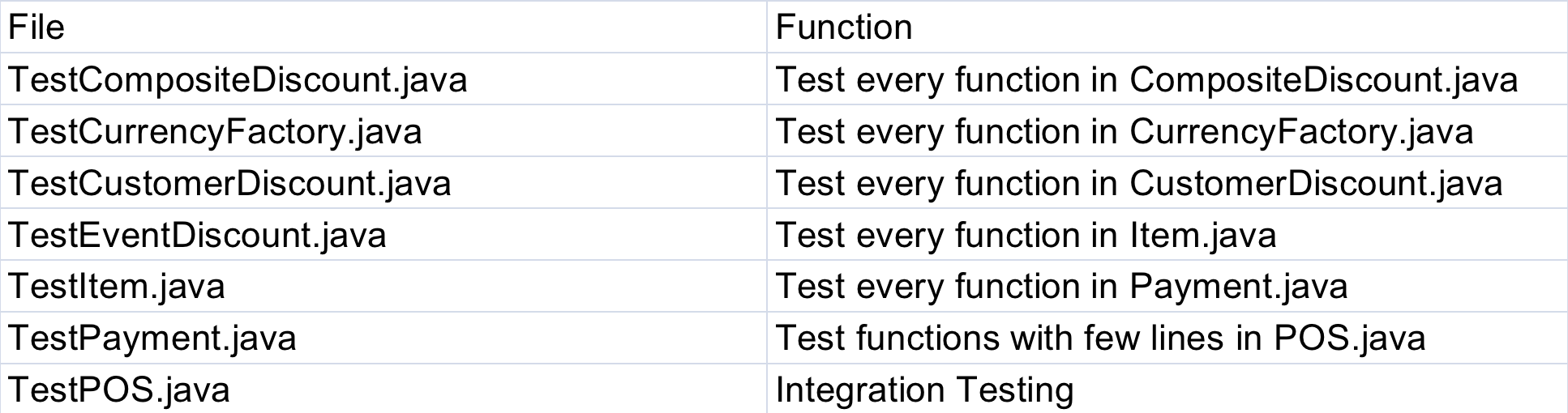


Fig 6. Detailed Branch Coverage

### **3.1 Test coverage each functional java file**

To achieve a high coverage for classes from this category is quite easy, just need to call all the member functions and construct functions. To get a coverage close to 100%, what need to be further considered is exception situation. For example, if we want to test Payment.java, we will need to set the all values(i.e., customer discount, holiday discount). Only in this way, we test all cases of Payment.java.

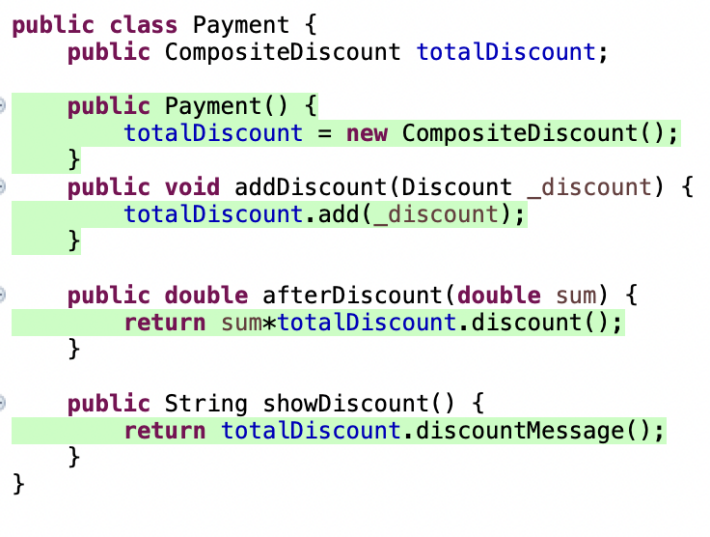
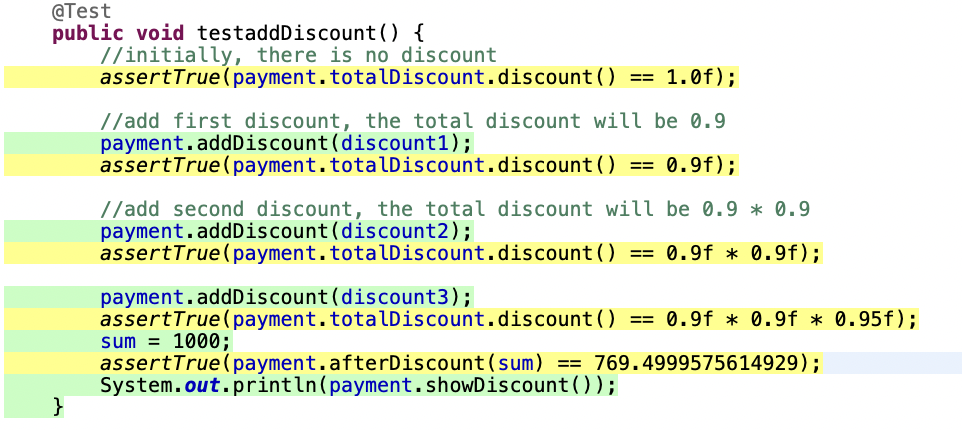


Fig 6. Payment.java

 Fig 7. test part of TestPayment.java

### Increasing coverage of classes related to POS main flow

To test these classes, what we need to do is simulating the purchase behavior under different situations in the command-line and BatchMode models, and achieve good branch coverage by setting up a reasonable test path to cover all program branches. In the TestMain.java file, I planned the integration test cases as follows to cover all the branches of POS.java as much as possible.

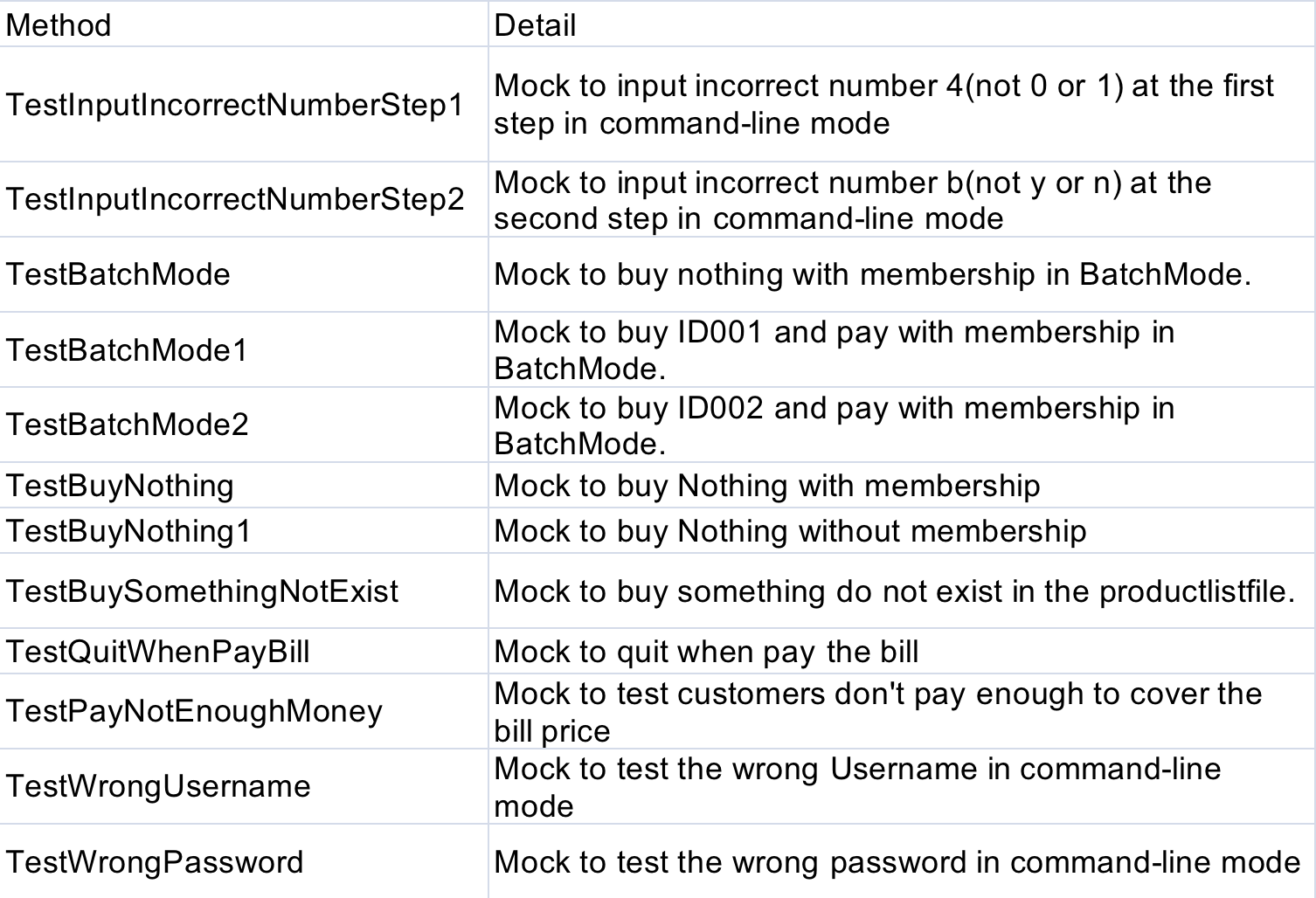


Fig 7. details about the methods of TestMain.java

## **Program statements that cannot be covered by unit tests**

(a) According to the FAQ of Assignment 1, the test cases do not need to cover those branches that unconditionally/always lead to the execution of System.exit. So I dismiss the lines and branches that lead to the System.exit(1). For example, There are many “System.exit(1)” for BatchMode in POS.java.

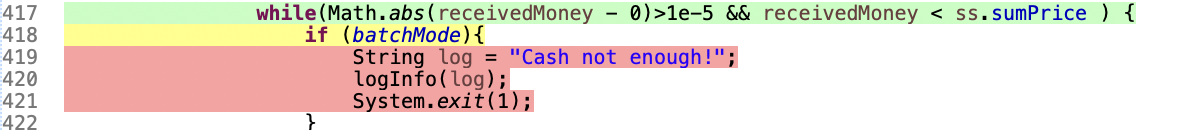


Fig 8. An example of Syetem,exit(1)

(b) In addition, there are cases where the subsequent java statements cannot be executed because System.exit(1) has been executed before. And here is an example.

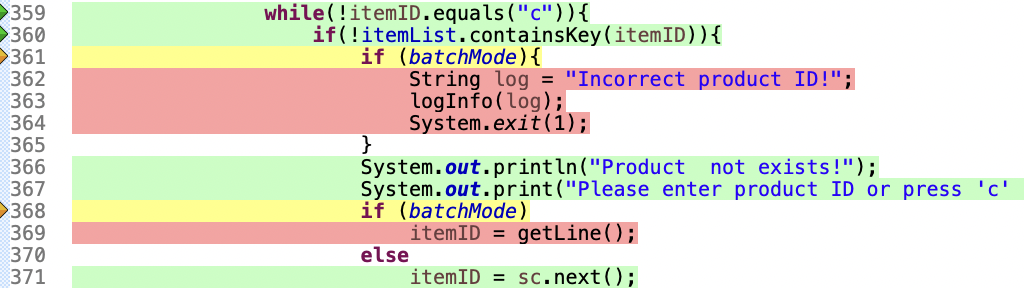


Fig 9. An example of cannot be executed due to the System.exit(1) execution

(c) The last case is the part of the statement that is not covered due to IOException. This is because there is no way to implement automatic IOException in the unit test, and IOException will not occur when the program runs normally, so this part of the code is not covered.

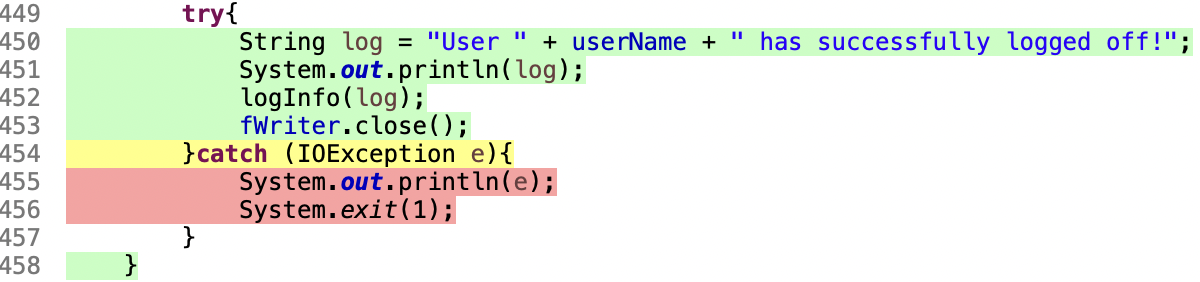


Fig 10. An example of cannot be executed due to IOException

## **Infeasible program statements**

This part is the bug I found in the program: due to the code in this part of the diagram below, there is no way to select whether there is a membership or not in the batchMode by setting “y” or “n” in the batchfile, but only by entering the command line of the console to select whether there is a membership.

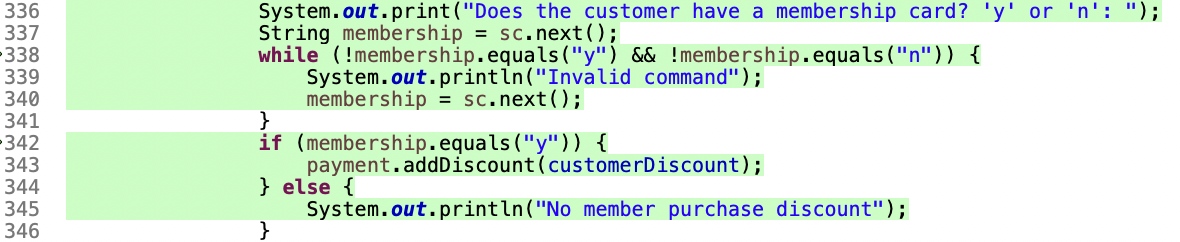


Fig 11. The part of codes that leads to the bug

## **Conclusion**

Although the system has some infeasible statements which cannot be tested, the test cases can still achieve high coverage. Also, it is necessary to simulate different situation such that different test flow are developed. At the same time, I also encountered some problems, such as the problem that the unit tests could not continue to be executed when the execution reached the test BatchMode, which is one of the reasons why my code coverage did not reach a very high.

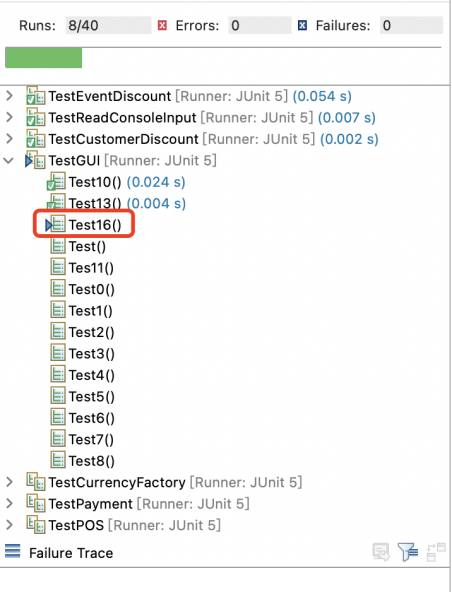




Fig 12. The problems confuse me in Assignment 1

## **Appendix**

Infeasible code in .java files:

In POS.java:

### **Because of reason (a) in Part 4:**

**Line 137 - Line 138:**

System.***out***.println("The user-password map file is wrong formatted!");

System.*exit*(1);

**Line 167 - Line 168:**

System.***out***.println("The item list file is wrongly formatted!");

System.*exit*(1);

**Line 253 - Line 256:**

**if** (*batchMode*){

String log = "Incorrect number!";

logInfo(log);

System.*exit*(1);

}

**Line 264 - Line 268:**

**if** (*batchMode*){

String log = "Incorrect number!";

logInfo(log);

System.*exit*(1);

}

**Line 286 - Line 290:**

**if** (*batchMode*){

String log = "Incorrect number!";

logInfo(log);

System.*exit*(1);

}

**Line 297 - Line 301:**

**if** (*batchMode*){

String log = "Incorrect number!";

logInfo(log);

System.*exit*(1);

}

**Line 328 - Line 330:**

**if** (*batchMode*)

System.*exit*(1);

}

**Line 418 - Line 422:**

**if** (*batchMode*){

String log = "Cash not enough!";

logInfo(log);

System.*exit*(1);

}

### **Because of reason (b) in Part 4:**

**Line 209 - Line 213:**

**if** (*batchMode*)

System.*exit*(1);

System.***out***.print("Please enter your user name:" );

**if** (*batchMode*)

userName = getLine();

**Line 228 - Line 232:**

**if** (*batchMode*)

System.*exit*(1);

System.***out***.print("Please enter your password: ");

**if** (*batchMode*)

password = getLine();

**Line 361 - Line 365 & Line 368 - Line 369:**

**if** (*batchMode*){

String log = "Incorrect product ID!";

logInfo(log);

System.*exit*(1);

}

**if** (*batchMode*)

itemID = getLine();

### **Because of reason (c) in Part 4:**

**Line 176 - Line 178:**

**catch**(IOException e){

System.***out***.println(e);

System.*exit*(1); // 1 means abnormal termination, 0 means normal termination

**Line 188 - Line 189:**

**catch**(IOException e) {

System.***out***.println(e);

System.*exit*(1);

}

**Line 454 - Line 457:**

**catch** (IOException e){

System.***out***.println(e);

System.*exit*(1);

}

**Line 476 - Line 479:**

**catch** (IOException e){

System.***out***.println(e);

System.*exit*(1);

}