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:

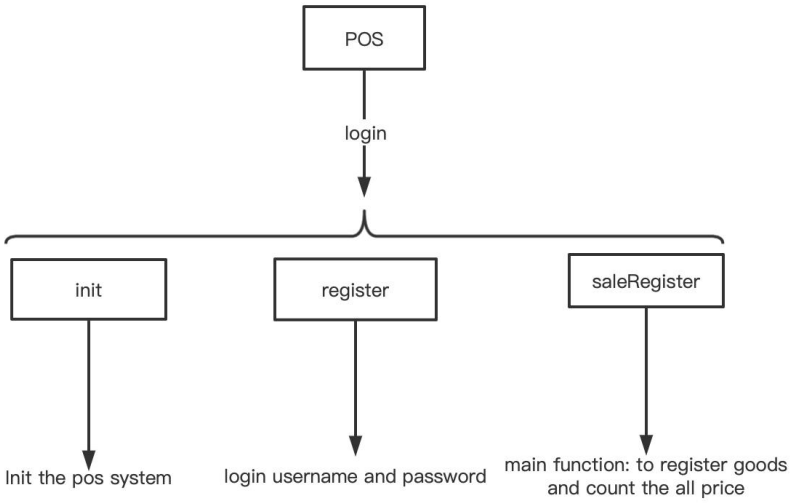

Fig 1. POS Flow Chart

2 Coverage

Overall coverage:

Element	Coverage	Covered Lines	Missed Lines	Total Lines
POS	92.2 %	676	57	733
src	88.6 %	327	42	369
(default package)	88.6 %	327	42	369
test	95.9 %	349	15	364

Fig 2. Line Coverage

Element	Coverage	Covered Branches	Missed Branches	Total Branches
POS	75.3 %	116	38	154
src	82.5 %	99	21	120
test	50.0 %	17	17	34

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:

Element	Coverage	Covered Lines	Missed Lines	Total Lines
POS	92.2 %	676	57	733
src	88.6 %	327	42	369
(default package)	88.6 %	327	42	369
POS.java	86.9 %	258	39	297
CurrencyFactory.java	85.7 %	6	1	7
TMNoTax.java	50.0 %	1	1	2
TMVAT.java	50.0 %	1	1	2
CompositeDiscount.java	100.0 %	15	0	15
Currency.java	100.0 %	1	0	1
CustomerDiscount.java	100.0 %	6	0	6
EventDiscount.java	100.0 %	6	0	6
HKCurrency.java	100.0 %	2	0	2
Item.java	100.0 %	8	0	8
Payment.java	100.0 %	7	0	7
ProductDiscount.java	100.0 %	6	0	6
SalesLine.java	100.0 %	8	0	8
USCurrency.java	100.0 %	2	0	2

Fig 4. Detailed Line Coverage

Element	Coverage	Covered Branches	Missed Branches	Total Branches
POS	75.3 %	116	38	154
src	82.5 %	99	21	120
(default package)	82.5 %	99	21	120
POS.java	81.2 %	91	21	112
CompositeDiscount.java	100.0 %	4	0	4
Currency.java		0	0	0
CurrencyFactory.java	100.0 %	4	0	4
CustomerDiscount.java		0	0	0
EventDiscount.java		0	0	0
HKCurrency.java		0	0	0
Item.java		0	0	0
Payment.java		0	0	0
ProductDiscount.java		0	0	0
SalesLine.java		0	0	0
TMNoTax.java		0	0	0
TMVAT.java		0	0	0
USCurrency.java		0	0	0

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.

File	Function
TestCompositeDiscount.java	Test every function in CompositeDiscount.java
TestCurrencyFactory.java	Test every function in CurrencyFactory.java
TestCustomerDiscount.java	Test every function in CustomerDiscount.java
TestEventDiscount.java	Test every function in Item.java
TestItem.java	Test every function in Payment.java
TestPayment.java	Test functions with few lines in POS.java
TestPOS.java	Integration Testing

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.

```
public class Payment {
    public CompositeDiscount totalDiscount;

    public Payment() {
        totalDiscount = new CompositeDiscount();
    }
    public void addDiscount(Discount _discount) {
        totalDiscount.add(_discount);
    }

    public double afterDiscount(double sum) {
        return sum*totalDiscount.discount();
    }

    public String showDiscount() {
        return totalDiscount.discountMessage();
    }
}
```

Fig 6. Payment.java

```
@Test
public void testaddDiscount() {
    //initially, there is no discount
    assertTrue(payment.totalDiscount.discount() == 1.0f);

    //add first discount, the total discount will be 0.9
    payment.addDiscount(discount1);
    assertTrue(payment.totalDiscount.discount() == 0.9f);

    //add second discount, the total discount will be 0.9 * 0.9
    payment.addDiscount(discount2);
    assertTrue(payment.totalDiscount.discount() == 0.9f * 0.9f);

    payment.addDiscount(discount3);
    assertTrue(payment.totalDiscount.discount() == 0.9f * 0.9f * 0.95f);
    sum = 1000;
    assertTrue(payment.afterDiscount(sum) == 769.4999575614929);
    System.out.println(payment.showDiscount());
}
```

Fig 7. test part of TestPayment.java

3.2 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.

Method	Detail
TestInputIncorrectNumberStep1	Mock to input incorrect number 4(not 0 or 1) at the first step in command-line mode
TestInputIncorrectNumberStep2	Mock to input incorrect number b(not y or n) at the second step in command-line mode
TestBatchMode	Mock to buy nothing with membership in BatchMode.
TestBatchMode1	Mock to buy ID001 and pay with membership in BatchMode.
TestBatchMode2	Mock to buy ID002 and pay with membership in BatchMode.
TestBuyNothing	Mock to buy Nothing with membership
TestBuyNothing1	Mock to buy Nothing without membership
TestBuySomethingNotExist	Mock to buy something do not exist in the productlistfile.
TestQuitWhenPayBill	Mock to quit when pay the bill
TestPayNotEnoughMoney	Mock to test customers don't pay enough to cover the bill price
TestWrongUsername	Mock to test the wrong Username in command-line mode
TestWrongPassword	Mock to test the wrong password in command-line mode

Fig 7. details about the methods of TestMain.java

4 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.

```
417                 while(Math.abs(receivedMoney - 0)>1e-5 && receivedMoney < ss.sumPrice ) {
418                     if (batchMode){
419                         String log = "Cash not enough!";
420                         logInfo(log);
421                         System.exit(1);
422                     }
```

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.

```
359                 while(!itemID.equals("c")){
360                     if(!itemList.containsKey(itemID)){
361                         if (batchMode){
362                             String log = "Incorrect product ID!";
363                             logInfo(log);
364                             System.exit(1);
365                         }
366                         System.out.println("Product  not exists!");
367                         System.out.print("Please enter product ID or press 'c'
368                         if (batchMode)
369                             itemID = getLine();
370                         else
371                             itemID = sc.next();
```

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.

```
449         try{
450             String log = "User " + userName + " has successfully logged off!";
451             System.out.println(log);
452             logInfo(log);
453             fWriter.close();
454         }catch (IOException e){
455             System.out.println(e);
456             System.exit(1);
457         }
458     }
```

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

5 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.

```
336                 System.out.print("Does the customer have a membership card? 'y' or 'n': ");
337                 String membership = sc.next();
338                 while (!membership.equals("y") && !membership.equals("n")) {
339                     System.out.println("Invalid command");
340                     membership = sc.next();
341                 }
342                 if (membership.equals("y")) {
343                     payment.addDiscount(customerDiscount);
344                 } else {
345                     System.out.println("No member purchase discount");
346                 }
```

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

6 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);
}
```