**Introduction**

This report outlines the unit testing brief for a point of sale called POS. The POS helps a customer to buy products and reserve the information of the buy process. This report consists of basic introduction of testing system, analysis of the testing procedures and the results.

**The Basic Structure of POS**



**Figure 1**: Basic Structure of POS

In the POS, Customers can buy items they want with legal currency and check the information about their payment in a log file. In general, POS project consists of four panels: Discount, Currency, Payment and POS.

In Discount panel, POS manger can make use of “*CompositeDiscount”* class to add or remove the discount of items on sale. There are three types of Discount: “*CustomerDiscoun”t*, “*EventDiscount”* and “*ProductDiscount”*. Each of the discount can be added or remove to the total discount which affects the total money customers need to pay. “*CustomerDiscount”* is valid only when the customer is a membership. “*EventDiscount”* and “*ProductDiscount”* is valid for all customers. Currency panel shows the legal currency in POS, which includes HK Dollar and US Dollar. Payment panel calculates the total money customers need to pay after the total discounts are considered. “*TaxModel”* should also be considered in Payment panel (while the POS never use it). POS panel is the main functional panel, customers can login POS with right username and password, then buy items with money in this panel. When the money is not enough, customers can cancel this payment. And “*SalesLine”* will return the information of all the items and money, and write the information in a log file.

**Test Procedure**

A suite of Junit test cases is constructed under Junit 5 framework. First, I test some independent class, for example: class “*Discount*”, class “*Currency*”*,* class “*Payment*”, to ensure that the methods in these class are correct. Then I test some static and initialized method in class “*POS*” with ”singleTest”. Finally, I test method register() in POS, including command from console and command from batch file.



**Figure 2**: Test Procedure

**Test Case Structure**

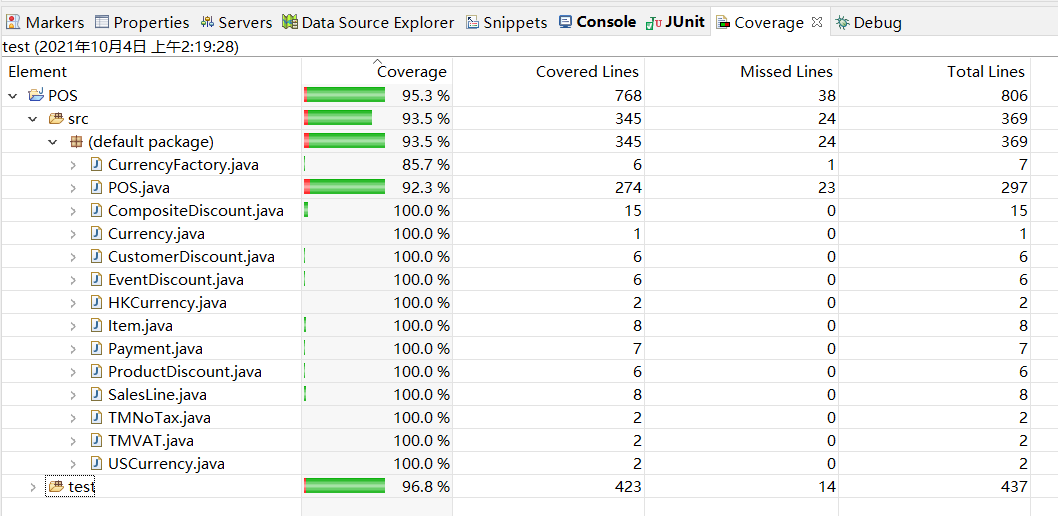


**Figure 3**: Test Case Structure

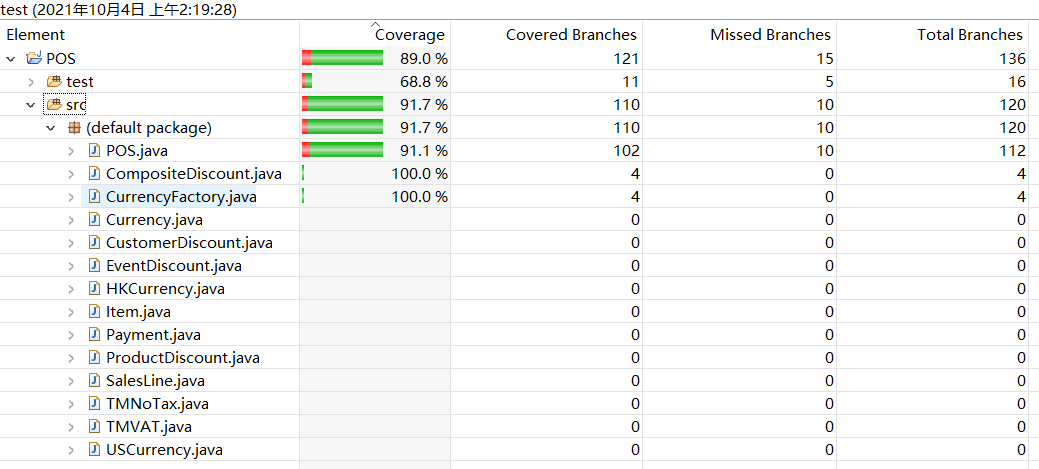
The Junit test cases for POS are represented by 14 test classes, and registered in the test suit “test”. In short, there is a one-to-one correspondence between each test class and each Java class of HMS except “POS” class. Considering there are two modes in the “POS” class, I divide the test into two parts: console and batch mode. What’s more, each method of each class of POS is tested by an individual test case of the corresponding test class.

**Statement Coverage and Brach Coverage(on Windows OS)**

The test cases can achieve 93.5% statement coverage, and 91.7% branch coverage for the source codes of POS.



**Figure 4**: Statement Coverage (on Windows OS)



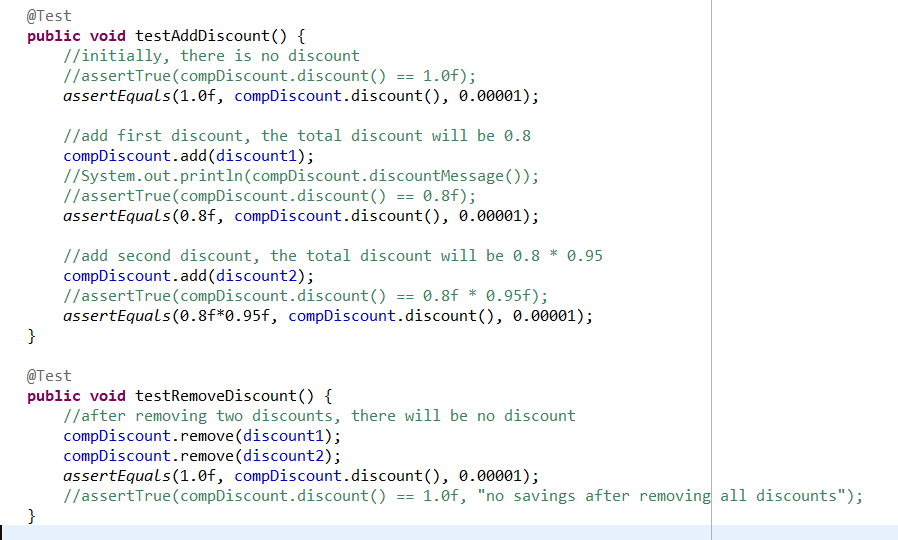
**Figure 5**: Branch Coverage (on Windows OS)

**High Code Coverage**

In order to get high code coverage, I divide the test in two parts: common used method and exceptions catch method. That is to say, besides the right commands, I also input some wrong commands to cover the “try catch” module and some.

1. Discount Panel
   1. TestCompositeDiscount

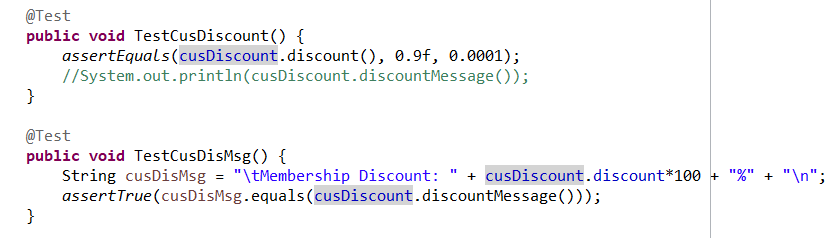
In order to test the correctness of function AddDiscount and RemoveDiscount, suppose some discount and calculate the right number of after/before discount, then compare them in Assert statement.



**Figure 6**: Some Test cases in “*TestCompositeDiscount*” class

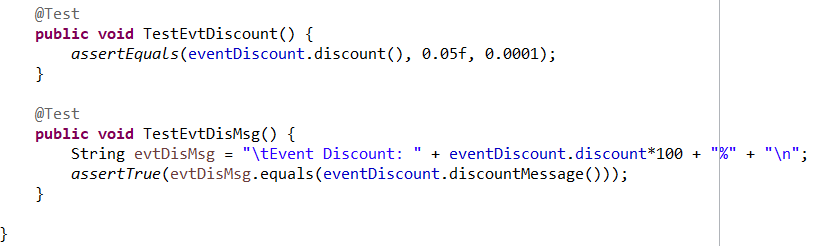
* 1. TestCustomerDiscount

To make sure that the manager can set the discount and get the right discount message return.



**Figure 7**: Some Test cases in “*TestCustomerDiscount*” class

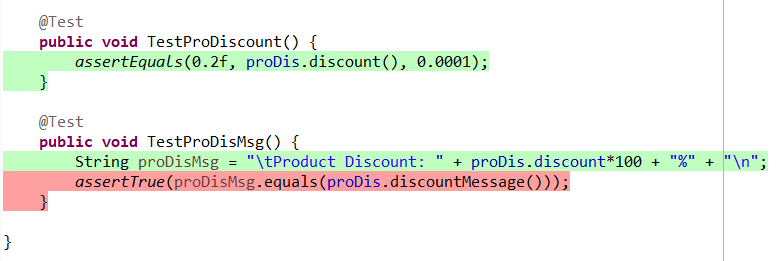
* 1. TestEventDiscount



**Figure 8**: Some Test cases in “*TestEventDiscount*” class

* 1. TestProductDiscount

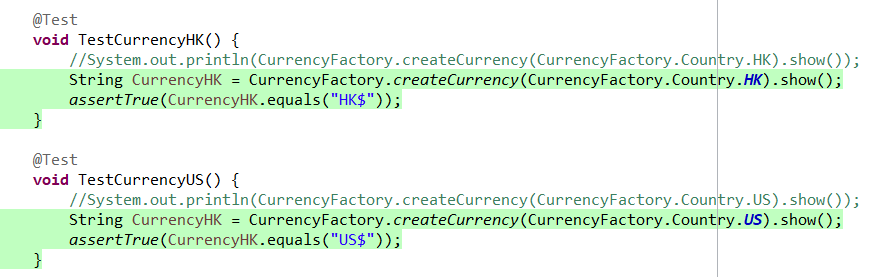
According to the Junit test result, there exists some fault in ProductDiscount of discountMessage: no discount number in this message.



**Figure 9**: Some Test cases in “*TestProductDiscount*” class

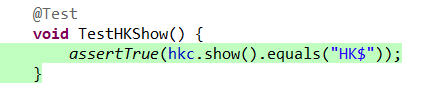
1. Currency Panel
   1. TestCurrencyFactory

Ensure that the function of creatCurrency works well to get corresponding currency.



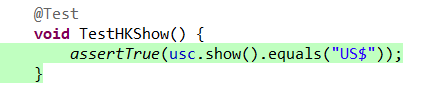
**Figure 10**: Some Test cases in “*TestCurrencyFactory*” class

* 1. TestCurrencyHK



**Figure 11**: Test case in “*TestCurrencyHK*” class

* 1. TestCurrencyUS



**Figure 12**: Test case of *TestHKShow()* in “*TestCurrencyUS*” class

1. Payment Panel
   1. TestPayment

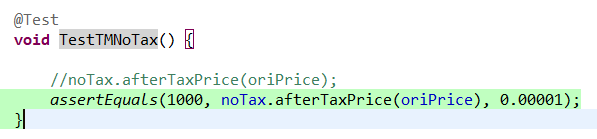


**Figure 13**: Some Test Cases of *AddDiscount()* in “*TestPayment*” class



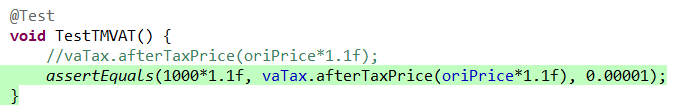
**Figure 14**: Some Test Cases of *AfterDiscount()*  in “*TestPayment*” class

* 1. TestTMNoTax



**Figure 15**: Test Case of *TMNOTax()*  in “*TMNOTax*” class

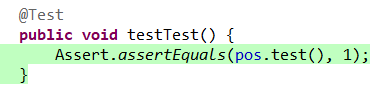
* 1. TestTMVAT



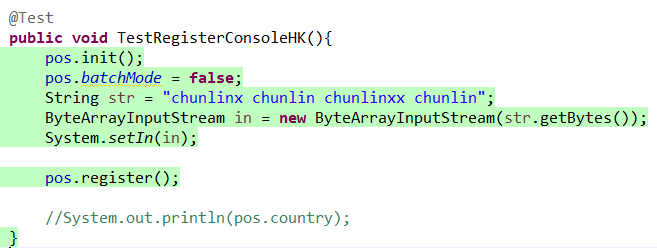
**Figure 16**: Test Case of *TMVAT()* in “*TestTMVAT*” class

1. POS Panel
   1. TestPOS

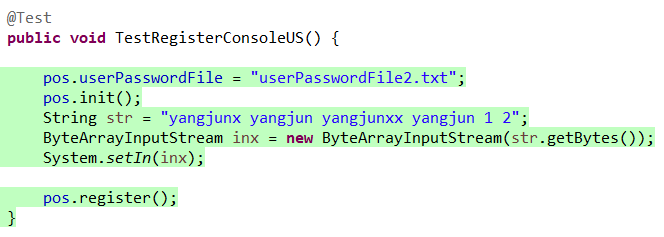
In this test, it’s an interactive mode. The POS accepts textual commands at console. In order to make the test get input from console, I change the input stream with the method “System.setIn()”, which means I set the input of console in one string then set this string as the input system of corresponding method. For example, in the method “regiter()”, we need to input the username and password in console, then I set the variable "str = ‘chunlinx chunlin chunlinxx chunlin’ “, and change the input stream. Thus the username and password is transmitted in “pos.register()”. If there is a need of start a new input stream, we can construct a extended class then override the corresponding method to change the input stream.



**Figure 17**: Test Case of *Test() in “TestPOS*” class



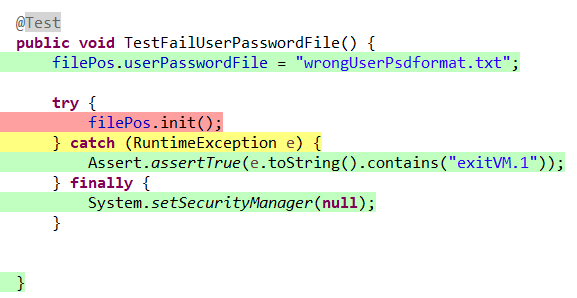
**Figure 18**: Test Case of *TestRegisterConsoleHK() in “TestPOS*” class



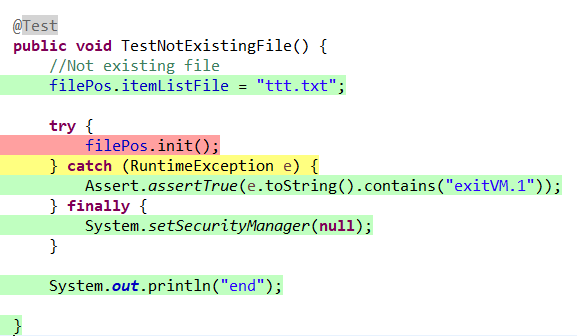
**Figure 19**: Test Case of *TestRegisterConsoleUS() in “TestPOS*” class

* 1. TestBatchMode

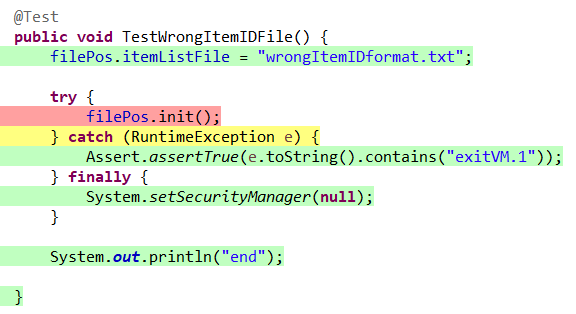
In this test mode, the POS accepts textual commands from a file. To cover the “try catch” module, it’s necessary to write some illegal input. For example, in the test case “TestFailUserPasswordFile”, the format of first line doesn’t match the “username password”, then it will lead exception. It’s similar for the situation that wrong commands are read from batch file. What’s more, because in this batch file test mode, when some exception happens, the test case execute statement “System.exit(1)”, the Junit will be terminated. As a result, I write many batch file for each type of illegal input.



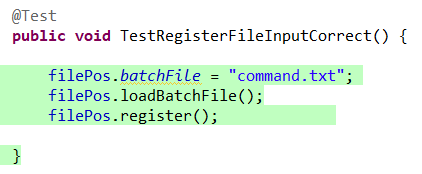
**Figure 20**: Test Case of *TestFailUserPasswordFile () in “TestBatchMode*” class



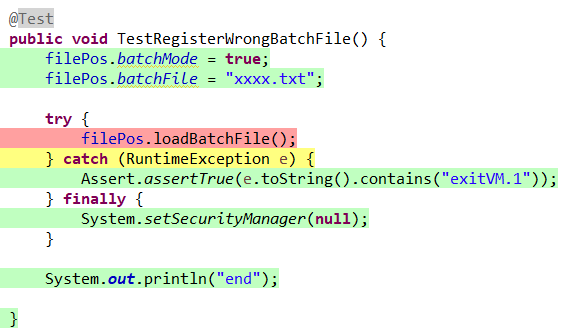
**Figure 21**: Test Case of *TestNotExistingFile () in “TestBatchMode*” class



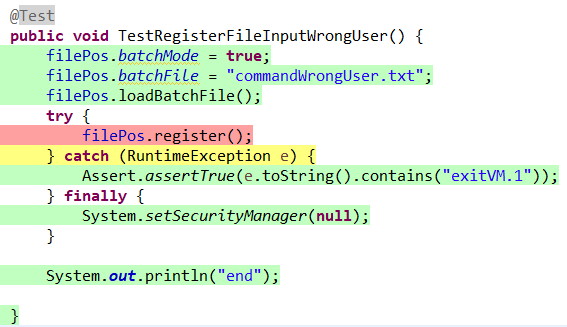
**Figure 22**: Test Case of *TestWrongItemFile () in “TestBatchMode*” class



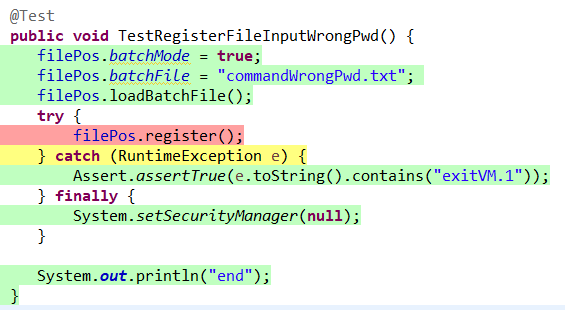
**Figure 23**: Test Case of *TestRegisterFileInput () in “TestBatchMode*” class



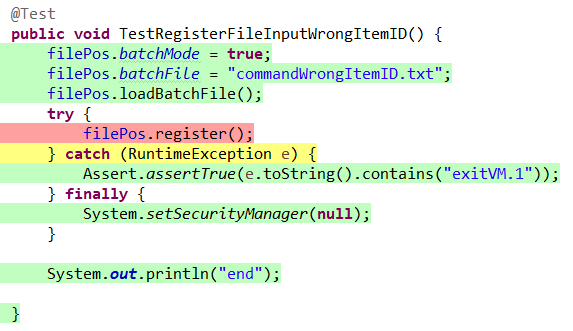
**Figure 24**: Test Case of *TestRegisterWrongBatchFile() in “TestBatchMode*” class



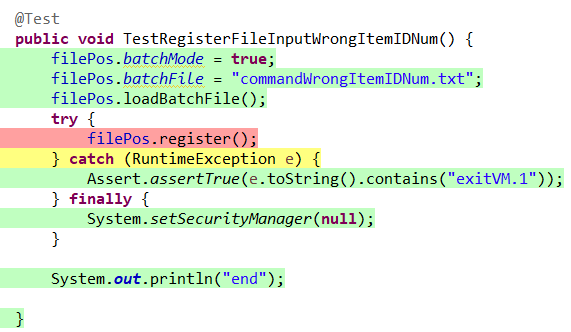
**Figure 25**: Test Case of *TestRegisterFileInputWrongUser () in “TestBatchMode*” class



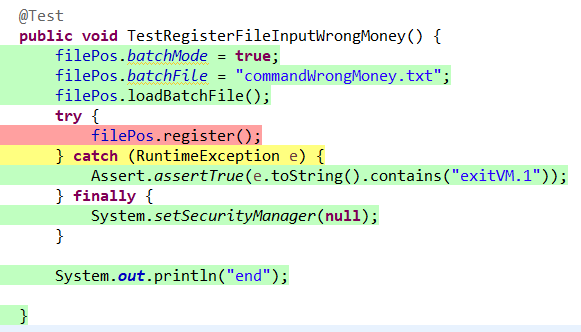
**Figure 26**: Test Case of *TestRegisterFileInputWrongPwd () in “TestBatchMode*” class



**Figure 27**: Test Case of *TestRegisterFileInputWrongItemID () in “TestBatchMode*” class



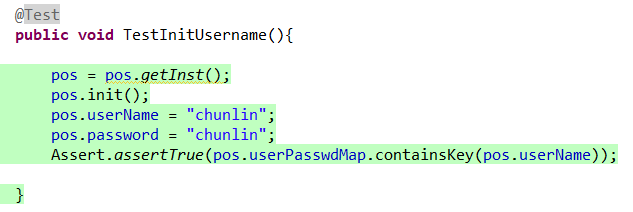
**Figure 28**: Test Case of *TestRegisterFileInputWrongItemIDNum() in “TestBatchMode*” class



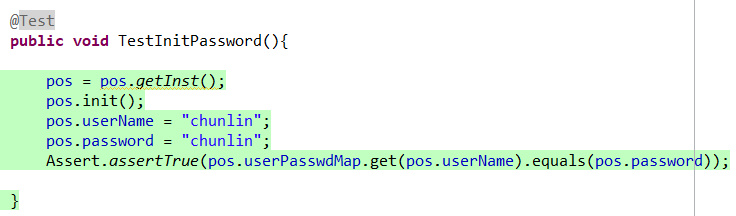
**Figure 29**: Test Case of *TestRegisterFileInputWrongMoney() in “TestBatchMode*” class

* 1. TestSingleTest

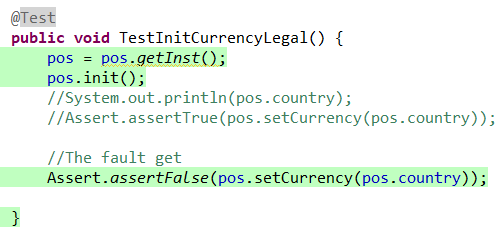
In this part, some static method, method without registration and variable initialization are tested in this test case. For example, I test the value of variables after method “init()” from class “POS”, including username, password, currency and so on.



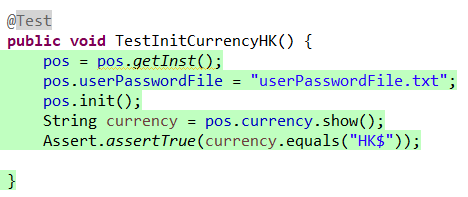
**Figure 30**: Test Case of *Init() in “TestSingleTest*” class



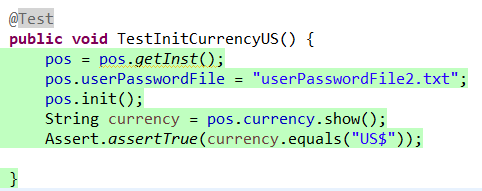
**Figure 31**: Test Case of *Init() in “TestSingleTest*” class



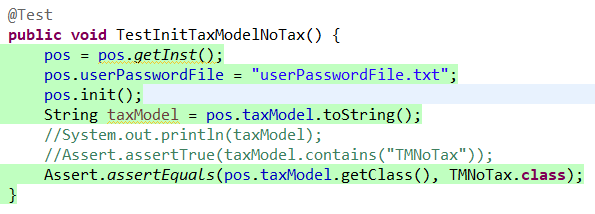
**Figure 32**: Test Case of *Init() in “TestSingleTest*” class



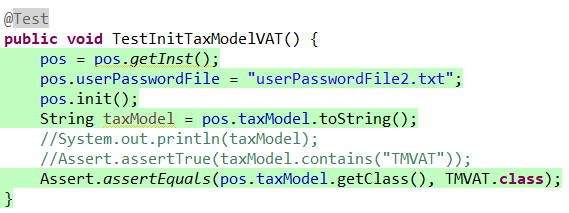
**Figure 33**: Test Case of *Init() in “TestSingleTest*” class



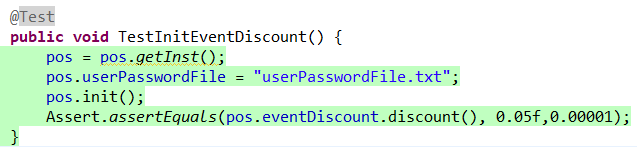
**Figure 34**: Test Case of *Init() in “TestSingleTest*” class



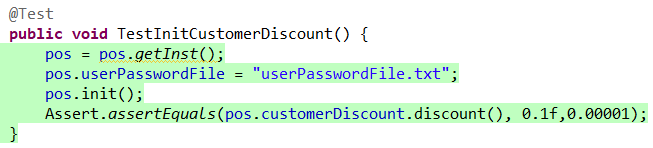
**Figure 35**: Test Case of *Init() in “TestSingleTest*” class



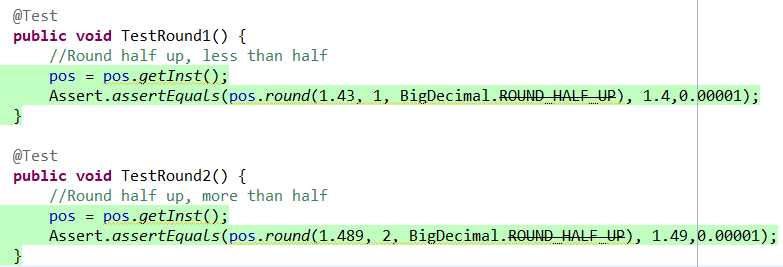
**Figure 36**: Test Case of *Init() in “TestSingleTest*” class



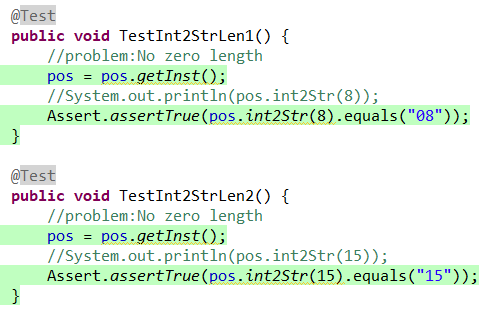
**Figure 37**: Test Case of *Init() in “TestSingleTest*” class



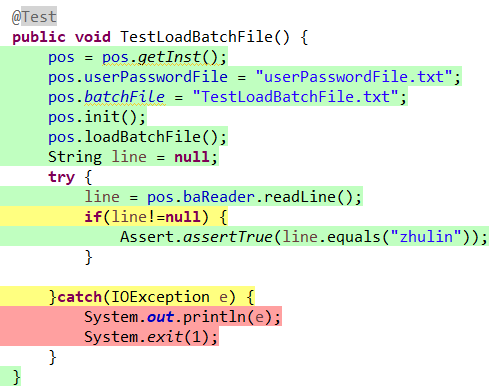
**Figure 38**: Test Case of *Init() in “TestSingleTest*” class



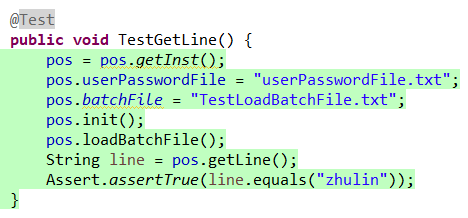
**Figure 39**: Some Test Cases of *round() in “TestSingleTest*” class



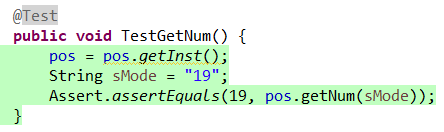
**Figure 40**: Some Test Cases of *int2Str() in “TestSingleTest*” class



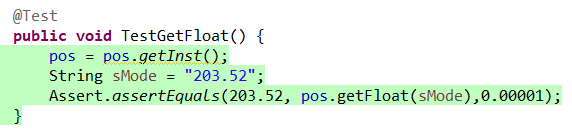
**Figure 41**: Test Case of *loadBatchFile () in “TestSingleTest*” class



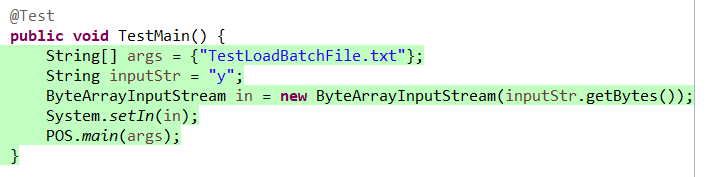
**Figure 42**: Test Case of *getLine() in “TestSingleTest*” class



**Figure 43**: Test Case of *getNum () in “TestSingleTest*” class

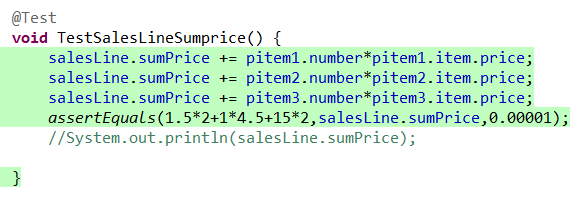


**Figure 44**: Test Case of *getFloat () in “TestSingleTest*” class



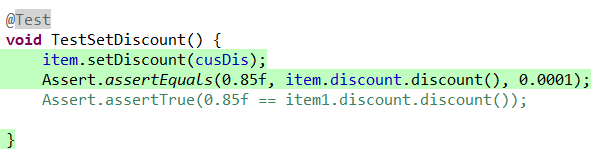
**Figure 45**: Test Case of *main() in “TestSingleTest*” class

* 1. TestSalesLine

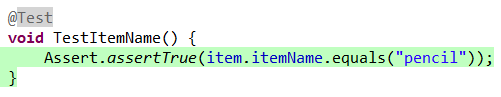


**Figure 46**: Test Case of *SalesLine() in “TestSalesLine*” class

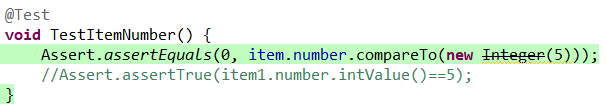
* 1. TestItem



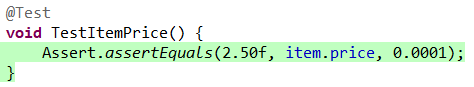
**Figure 47**: Test Case of *setDiscount() in “TestItem*” class



**Figure 48**: Test Case of *Item() in “TestItem*” class



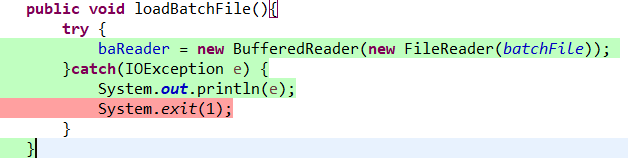
**Figure 49**: Test Case of *Item() in “TestItem*” class



**Figure 50**: Test Case of *Item() in “TestItem*” class

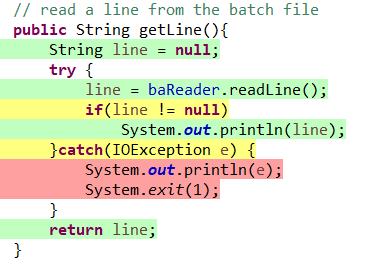
**Infeasible statement**

1. All of the sentence in method that describe “System.exit(1)”, for example, in the method loadBatchFile(). Because in testing stage, if it comes to the sentence “System.exit(1)”, “System.exit(1)” will terminates the currently running Java Virtual Machine, which means if test case execute statement “System.exit(1)”, the Junit will be terminated. Thus, the statement “System.exit(1)” cannot execute in any test case.



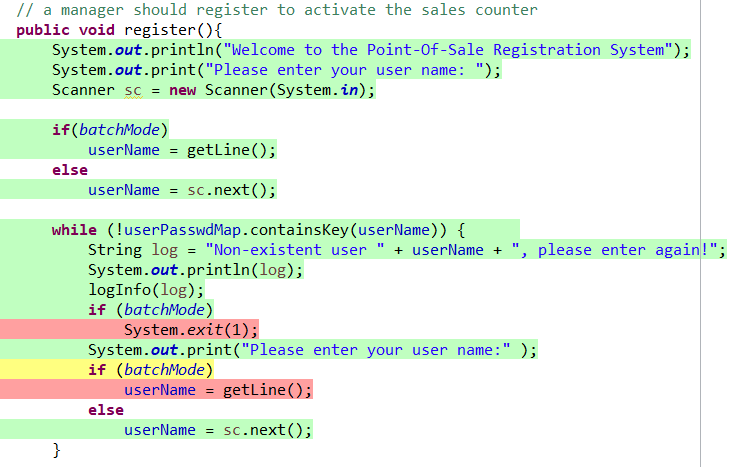
**Figure 51**: statement *“System.exit(1)” in “POS*” class

1. There is an infeasible statement in method getLine() which within the “POS” class. For example, if variable “line” is equal to null, which means some exception has happened then it will go catch module; otherwise it will in try module. That is to say in the try module, the variable “line” never be null.



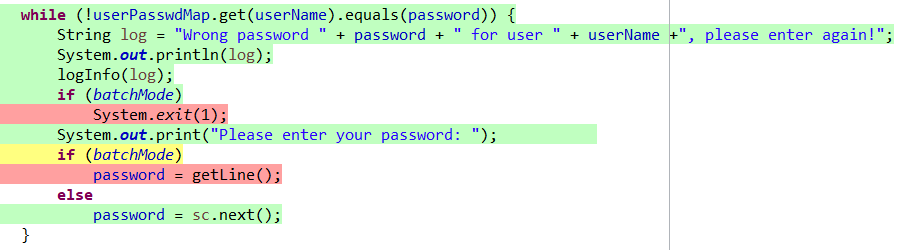
**Figure 52**: method *getline() in “POS*” class

1. There is an infeasible statement in method register() of “POS” class. As I mentioned in the first infeasible statement, “System.exit(1)” will make Junit terminate. And if the variable “batchMode” is true, it means in the batch mode, the POS accepts a nonexistent username from a file, then the “System.exit(1)” will be executed, the JUnit will be terminated, then it won’t comes to the next “if (batchMode)” branch. Thus, the statement “if(batchMode) username = getLine()” is an infeasible statement.



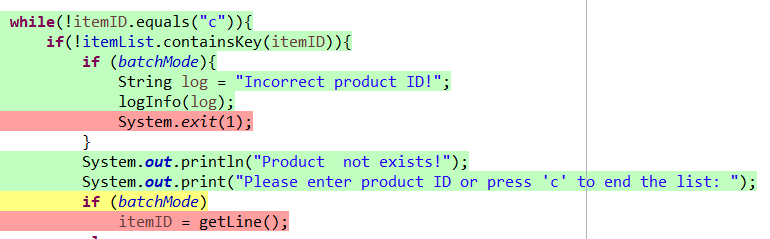
**Figure 53**: method *register() in “POS*” class

1. The situation is similar to the third infeasible statement. There is an infeasible statement in method register() of “POS” class. If the variable “batchMode” is true, it means in the batch mode, the POS accepts a wrong password for username from a file, then the “System.exit(1)” will be executed, the JUnit will be terminated, then it won’t comes to the next “if (batchMode)” branch. Thus, the statement “if(batchMode) password = getLine()” is an infeasible statement.



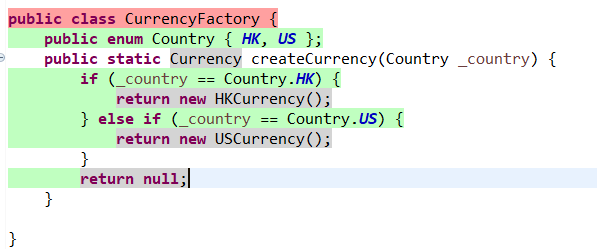
**Figure 54**: method *register() in “POS*” class

1. The situation is similar to the third and fourth infeasible statement. There is an infeasible statement in method register() of “POS” class. If the variable “batchMode” is true, it means in the batch mode, the POS accepts a nonexistent itemID from a file, then the “System.exit(1)” will be executed, the JUnit will be terminated, then it won’t comes to the “if (batchMode)” branch. Thus, the statement “if(batchMode) itemID = getLine()” is an infeasible statement.



**Figure 55**: method sale*Register() in “POS*” class

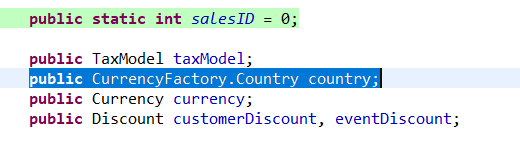
1. There is an infeasible statement in the construct method of “CurrencyFactory” class. The variable “CurrencyFactory” never be used in the project, which means there is no “new CurrencyFactory()” sentence. So the statement of “public class CurrencyFactory” is an infeasible statement.



**Figure 56**: method *CurrencyFactory() in “CurrencyFactory*” class

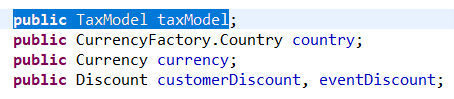
**The Program Failure**

1. The variable “public CurrencyFactory.Country country” in class “POS” is never used in the POS project.



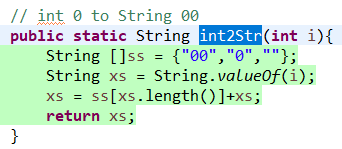
**Figure 57**: failure *in “POS*” class

1. The variable “public TaxModel taxModel” in class “POS” is never used in the POS project.



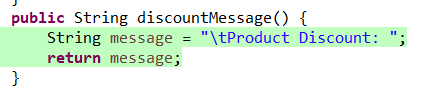
**Figure 58**: failure *in “POS*” class

1. It’s possible that saleID greater than 99, while the method “int2Str(int i)” can only deal with number 0~99. What’s more, in the “int2Str(int i)” method, the item “ss[0]” won’t be used because there is no integer’s length can be 0.



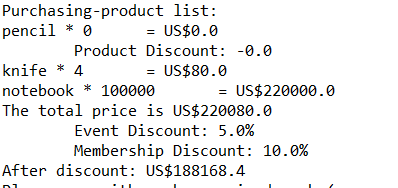
**Figure 59**: method *int2Str() in “POS*” class

1. There is no valid discount message in method “discountMessage()” in class “ProductDiscount”. The discount of the product doesn’t be added to the discount message string.



**Figure 60**: method *discountMessage() in “ProductDiscount*” class

1. The POS doesn’t check the existing number of corresponding items that customers buy. For example, if the storage of the notebook is only five, the customer can buy 10000 pencil successfully.



**Figure 61**: failure *in “POS*” class

**Conclusion**

Overall, the test cases for POS have achieved a high code coverage. The test cases have achieved 93.5% statement coverage, and 91.7% branch coverage which is higher than expected.