Assignment 3 Final Report - Electricity Discount Calculator Testing

Haioum David

May 10th 2020

Contents

1 Test strategy **1**

1.1 White box testing - Unit tests 1

1.2 Black box testing - Integration tests 1

2 White box testing - Unit tests **2**

2.1 Test design 2

2.2 Test Script 5

2.3 Test class (Java implementation) 6

3 Black box testing - Integration tests **7**

3.1 Test design 7

3.2 Test Script 7

3.3 Test class (Java implementation) 7

4 Coverage information **8**

4.1 Coverage report 8

4.2 Coverage optimisation 10

5 Conclusion **11**

1 Test strategy

In this section, I will discuss on the Test strategy of the given electricity discount calculator code. Which means that after analyzing the software source code and how it works once launched, the goal is to provide the best strategy to conduct the tests.

I will start by explaining my unit-testing strategy with well-known White Box testing technics. Then I will continue to explain how I conducted the integration tests on the main program by using black box testing technics. This strategy aims to follow the V-cycle model[1].

1.1 White box testing - Unit tests

We would like that every software components and functions work individually by examining each line of codes and its data structure. It will help to optimizing the code and determine the location of any problem. The specification document is given and the implementation of the code don't seem complex.

Therefore, the chosen solution to conduct unit tests is Basic Path Testing method. The flow graphs are already given in the specification document, we can easily generate the flow charts to extract the cyclomatic complexity $V(G)$ and the independent paths to determine how many test cases we need to satisfy a good coverage percentage.

By continue reading this report you will find the result of this method and whether it was effective or not.

1.2 Black box testing - Integration tests

For the integration test, we supose that the source code is unknown and we don't have any knowledge on the internal structure. This approach will be based on the legal inputs and the expected outputs of the program.

The chosen solutions are Decision Table testing and Boundary Value Analysis (BVA). The decision table is a matrix representation of the logic of a decision. This table is already given in the specification table and already simplified.

In addition, BVA complements the Decision table. We will then focus on the bundaries of the input and output values. After analyzing the specifications, we can see that the electricty discount calculation is based on ranges (example: comsuption is included between 0 kWh and 200 kWh). Then it is important to test the bundaries of this ranges.

[1] A graphical representation of a systems development lifecycle

2 White box testing - Unit tests

In this section let's have a closer look on the design and implementation of the unit tests

2.1 Test design

All the control flow charts have been generated by analyzing the source code and the control flow graphs from the specification document.

2.1.1 calcTax

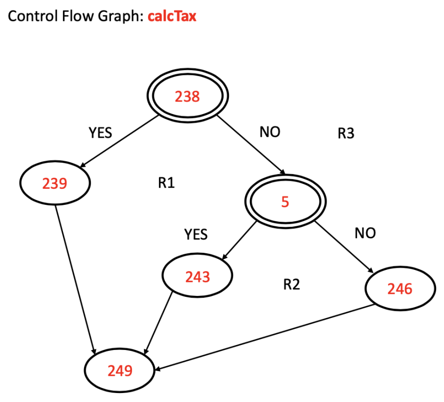


Figure 1: *calcTax* control flow graph

$$P = 7$$

$$N = 6$$

$$V(G) = P - N + 2 = 3$$

3 regions counted and $V(G) = 3$ then **3 test cases** generated.

2.1.2 calcDiscBilNR1 and calcDiscBilNR2

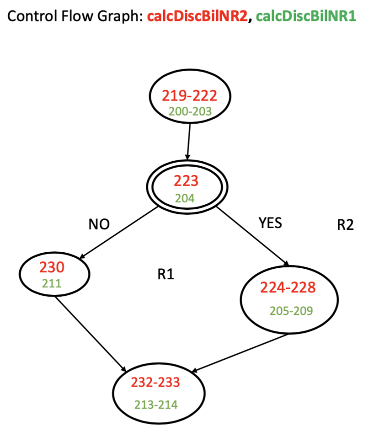


Figure 2: *calcDiscBilNR1* and calcDiscBilNR2 control flow graph

$$P = 5$$

$$N = 5$$

$$V(G) = P - N + 2 = 2$$

2 regions counted and $V(G) = 2$ then **2 test cases** generated.

2.1.3 calcNoDiscBilNR and calcNoDiscBilR

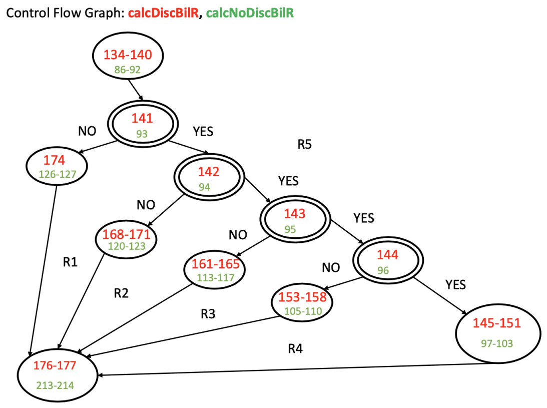


Figure 3: *calcNoDiscBilNR* and calcNoDiscBilR control flow graph

$$P = 14$$

$$N = 11$$

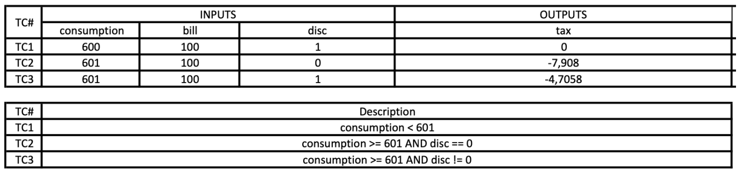
$$V(G) = P - N + 2 = 5$$

5 regions counted and $V(G) = 5$ then **5 test cases** generated.

2.2 Test Script

Let's generate the test cases from the control flow charts in the section 2.1.

2.2.1 calcTax function

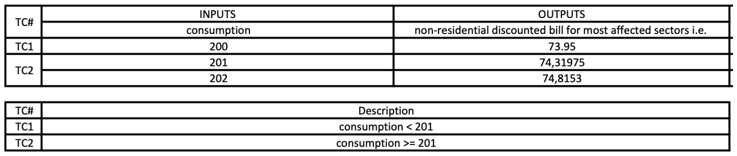


TC#	INPUTS			OUTPUTS
	consumption	bill	disc	tax
TC1	600	100	1	0
TC2	601	100	0	-7,908
TC3	601	100	1	-4,7058

TC#	Description
TC1	consumption < 601
TC2	consumption >= 601 AND disc == 0
TC3	consumption >= 601 AND disc != 0

Figure 4: *calcTax* function test cases

2.2.2 calcDiscBilNR1 function

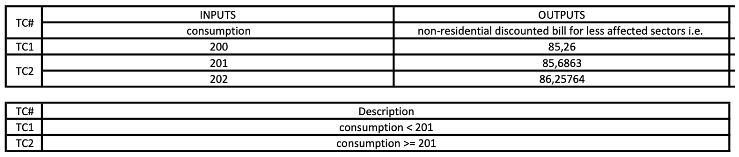


TC#	INPUTS	OUTPUTS
	consumption	non-residential discounted bill for most affected sectors i.e.
TC1	200	73.95
TC2	201	74,31975
	202	74,8153

TC#	Description
TC1	consumption < 201
TC2	consumption >= 201

Figure 5: *calcDiscBilNR1* function test cases

2.2.3 calcDiscBilNR2 function

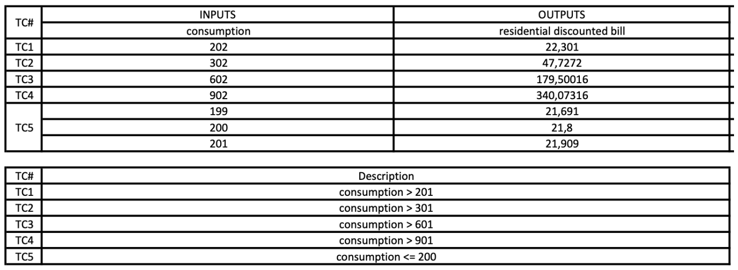


TC#	INPUTS	OUTPUTS
	consumption	non-residential discounted bill for less affected sectors i.e.
TC1	200	85,26
TC2	201	85,6863
	202	86,25764

TC#	Description
TC1	consumption < 201
TC2	consumption >= 201

Figure 6: *calcDiscBilNR2* function test cases

2.2.4 calcDiscBilR function

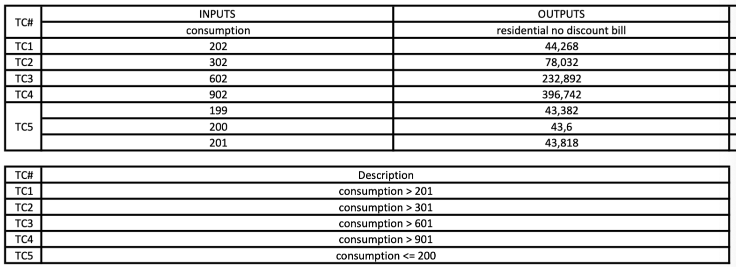


TC#	INPUTS	OUTPUTS
	consumption	residential discounted bill
TC1	202	22,301
TC2	302	47,7272
TC3	602	179,50016
TC4	902	340,07316
TC5	199	21,691
	200	21,8
	201	21,909

TC#	Description
TC1	consumption > 201
TC2	consumption > 301
TC3	consumption > 601
TC4	consumption > 901
TC5	consumption <= 200

Figure 7: *calcDiscBilR* function test cases

2.2.5 calcNoDiscBilR function

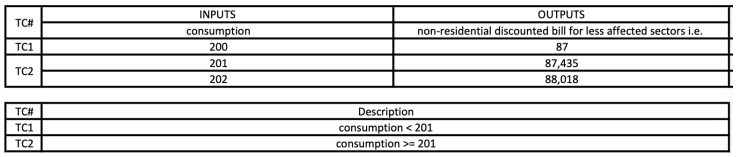


TC#	INPUTS	OUTPUTS
	consumption	residential no discount bill
TC1	202	44,268
TC2	302	78,032
TC3	602	232,892
TC4	902	396,742
TC5	199	43,382
	200	43,6
	201	43,818

TC#	Description
TC1	consumption > 201
TC2	consumption > 301
TC3	consumption > 601
TC4	consumption > 901
TC5	consumption <= 200

Figure 8: *calcNoDiscBilR* function test cases

2.2.6 calcNoDiscBilNR function



TC#	INPUTS	OUTPUTS
	consumption	non-residential discounted bill for less affected sectors i.e.
TC1	200	87
TC2	201	87,435
	202	88,018

TC#	Description
TC1	consumption < 201
TC2	consumption >= 201

Figure 9: *calcNoDiscBilNR* function test cases

For more details, see the ***unit_tests_cfg.pdf*** document which give you the inputs and outputs of each test cases concerning unit tests only.

2.3 Test class (Java implementation)

You can see the code in the file ***ElectricityDiscountCalcTest.java***

3 Black box testing - Integration tests

3.1 Test design

The test script has been generated from the given decision table and simplified.

The first idea was to generate all the test cases from the given decision table. But I realized the output of the program was not the same. In the final test script we want to test the 7 outputs displayed in the terminal from the the 4 inputs provided by the user.

We consider that the input *continue* is always 0 to perform only one time the program for each test case. Then in the final test script, I simplified the last cases for *Hotel*, *Travel*, *Commercial*, *Industrial*. We will see later that this optimisation have an effect on the coverage (Figure 15).

3.2 Test Script

For more details, see the ***integration_tests_script.pdf*** document which give you the inputs and outputs of each test cases concerning unit tests only.

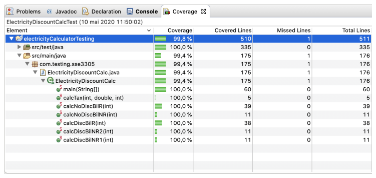
3.3 Test class (Java implementation)

You can see the code in the file ***ElectricityDiscountCalcIT.java*** which is nested in ***ElectricityDiscountCalcTest.java***.

4 Coverage information

4.1 Coverage report

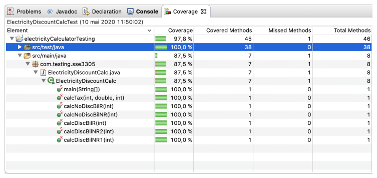
This section present you the results (Figure 10) of the tests known as coverage that give the percentages. Figure 11 illustrate why certain coverage percentages are not at 100%. This coverage report has been made on the test script with simplifications mentioned in Section 3.1



ElectricityDiscountCalcTest (10 mai 2020 11:50:02)

Element	Coverage	Covered Lines	Missed Lines	Total Lines
electricityCalculatorTesting	99,8 %	510	1	511
src/test/java	100,0 %	335	0	335
src/main/java	99,4 %	175	1	176
com.testing.sse3305	99,4 %	175	1	176
ElectricityDiscountCalc.java	99,4 %	175	1	176
ElectricityDiscountCalc	99,4 %	175	1	176
main(String[])	100,0 %	60	0	60
calcTax(int, double, int)	100,0 %	5	0	5
calcNoDiscBilR(int)	100,0 %	39	0	39
calcNoDiscBilNR(int)	100,0 %	11	0	11
calcDiscBilR(int)	100,0 %	38	0	38
calcDiscBilNR2(int)	100,0 %	11	0	11
calcDiscBilNR1(int)	100,0 %	11	0	11

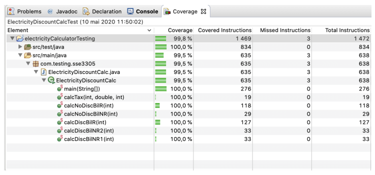
(a) Line coverage



ElectricityDiscountCalcTest (10 mai 2020 11:50:02)

Element	Coverage	Covered Methods	Missed Methods	Total Methods
electricityCalculatorTesting	97,8 %	45	1	46
src/test/java	100,0 %	38	0	38
src/main/java	87,5 %	7	1	8
com.testing.sse3305	87,5 %	7	1	8
ElectricityDiscountCalc.java	87,5 %	7	1	8
ElectricityDiscountCalc	87,5 %	7	1	8
main(String[])	100,0 %	1	0	1
calcTax(int, double, int)	100,0 %	1	0	1
calcNoDiscBilR(int)	100,0 %	1	0	1
calcNoDiscBilNR(int)	100,0 %	1	0	1
calcDiscBilR(int)	100,0 %	1	0	1
calcDiscBilNR2(int)	100,0 %	1	0	1
calcDiscBilNR1(int)	100,0 %	1	0	1

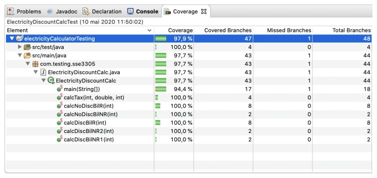
(b) Method coverage



ElectricityDiscountCalcTest (10 mai 2020 11:50:02)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
electricityCalculatorTesting	99,8 %	1 469	3	1 472
src/test/java	100,0 %	834	0	834
src/main/java	99,5 %	635	3	638
com.testing.sse3305	99,5 %	635	3	638
ElectricityDiscountCalc.java	99,5 %	635	3	638
ElectricityDiscountCalc	99,5 %	635	3	638
main(String[])	100,0 %	276	0	276
calcTax(int, double, int)	100,0 %	19	0	19
calcNoDiscBilR(int)	100,0 %	118	0	118
calcNoDiscBilNR(int)	100,0 %	29	0	29
calcDiscBilR(int)	100,0 %	127	0	127
calcDiscBilNR2(int)	100,0 %	33	0	33
calcDiscBilNR1(int)	100,0 %	33	0	33

(c) Instruction coverage



ElectricityDiscountCalcTest (10 mai 2020 11:50:02)

Element	Coverage	Covered Branches	Missed Branches	Total Branches
electricityCalculatorTesting	97,9 %	47	1	48
src/test/java	100,0 %	4	0	4
src/main/java	97,7 %	43	1	44
com.testing.sse3305	97,7 %	43	1	44
ElectricityDiscountCalc.java	97,7 %	43	1	44
ElectricityDiscountCalc	97,7 %	43	1	44
main(String[])	94,4 %	17	1	18
calcTax(int, double, int)	100,0 %	4	0	4
calcNoDiscBilR(int)	100,0 %	8	0	8
calcNoDiscBilNR(int)	100,0 %	2	0	2
calcDiscBilR(int)	100,0 %	8	0	8
calcDiscBilNR2(int)	100,0 %	2	0	2
calcDiscBilNR1(int)	100,0 %	2	0	2

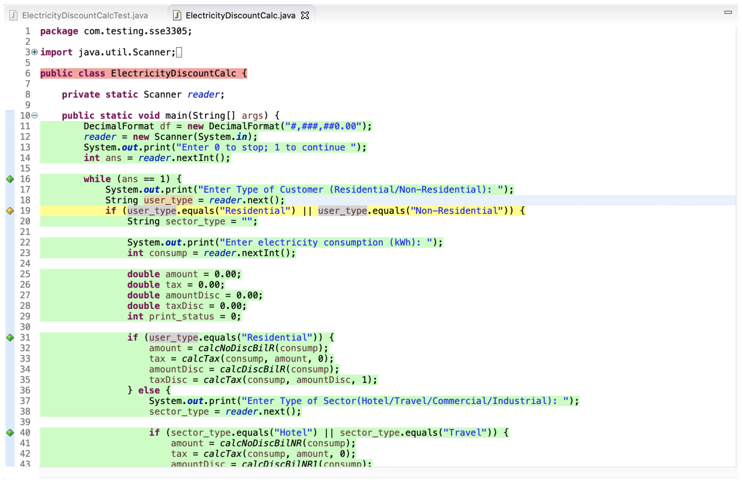
(d) Branch coverage

Figure 10: Test coverage report from Jacoco Eclipse

On Figure 10 - (c), we can see that the tests provide a 100% coverage on every functions. Which means our tests are efficient in term of Instruction coverage (all instructions has been tested).

For line coverage (Figure 10 - (a)) as well: The tests sucessfully cover all the lines of code.

However we can see that the Method coverage (Figure 10 - (b) and Branch coverage (Figure 10 - (d)) are not completely done. Note that even if we said that line coverage provide a 100% coverage we can notice that the total coverage is 99.4%. On the Figure 11 we can have a look on what is really happening and try to find a suitable solution.



ElectricityDiscountCalcTest.java | ElectricityDiscountCalc.java

```
1  package com.testing.sse3305;
2
3  import java.util.Scanner;
5
6  public class ElectricityDiscountCalc {
7
8      private static Scanner reader;
9
10     public static void main(String[] args) {
11         DecimalFormat df = new DecimalFormat("#,###,##0.00");
12         reader = new Scanner(System.in);
13         System.out.print("Enter 0 to stop; 1 to continue ");
14         int ans = reader.nextInt();
15
16         while (ans == 1) {
17             System.out.print("Enter Type of Customer (Residential/Non-Residential): ");
18             String user_type = reader.next();
19             if (user_type.equals("Residential") || user_type.equals("Non-Residential")) {
20                 String sector_type = "";
21
22                 System.out.print("Enter electricity consumption (kWh): ");
23                 int consump = reader.nextInt();
24
25                 double amount = 0.00;
26                 double tax = 0.00;
27                 double amountDisc = 0.00;
28                 double taxDisc = 0.00;
29                 int print_status = 0;
30
31                 if (user_type.equals("Residential")) {
32                     amount = calcNoDiscBilR(consump);
33                     tax = calcTax(consump, amount, 0);
34                     amountDisc = calcDiscBilR(consump);
35                     taxDisc = calcTax(consump, amountDisc, 1);
36                 } else {
37                     System.out.print("Enter Type of Sector(Hotel/Travel/Commercial/Industrial): ");
38                     sector_type = reader.next();
39
40                     if (sector_type.equals("Hotel") || sector_type.equals("Travel")) {
41                         amount = calcNoDiscBilNR(consump);
42                         tax = calcTax(consump, amount, 0);
43                         amountDisc = calcDiscBilNR1(consump);
```

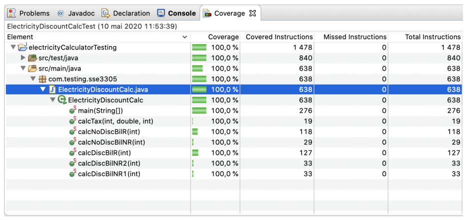
Figure 11: The coverage of the ElectricityDiscountCalc java class

On Figure 11 at line 6 of the program, The instanciation of the class is not really tested. This impact the coverage percentage. We could not have predict this case in our test design. So let's write a test that test the instanciation in java (Figure 12).

```
@Test
public void classShouldInstantiateMainClass() {
    assertTrue((new ElectricityDiscountCalc()) instanceof ElectricityDiscountCalc);
}
```

Figure 12: Verify the instance of ElectricityDiscountCalc java class

Now we can launch again the coverage on eclipse and observe the results.

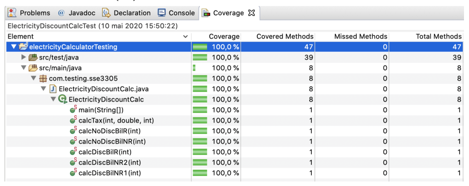


Problems | Javadoc | Declaration | Console | Coverage

ElectricityDiscountCalcTest (10 mai 2020 11:53:39)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
electricityCalculatorTesting	100,0 %	1 478	0	1 478
src/test/java	100,0 %	840	0	840
src/main/java	100,0 %	638	0	638
com.testing.sse3305	100,0 %	638	0	638
ElectricityDiscountCalc.java	100,0 %	638	0	638
ElectricityDiscountCalc	100,0 %	638	0	638
main(String[])	100,0 %	276	0	276
calcTax(int, double, int)	100,0 %	19	0	19
calcNoDiscBilR(int)	100,0 %	118	0	118
calcNoDiscBilNR(int)	100,0 %	29	0	29
calcDiscBilR(int)	100,0 %	127	0	127
calcDiscBilNR2(int)	100,0 %	33	0	33
calcDiscBilNR1(int)	100,0 %	33	0	33

(a) Instruction coverage



Problems | Javadoc | Declaration | Console | Coverage

ElectricityDiscountCalcTest (10 mai 2020 15:50:22)

Element	Coverage	Covered Methods	Missed Methods	Total Methods
electricityCalculatorTesting	100,0 %	47	0	47
src/test/java	100,0 %	39	0	39
src/main/java	100,0 %	8	0	8
com.testing.sse3305	100,0 %	8	0	8
ElectricityDiscountCalc.java	100,0 %	8	0	8
ElectricityDiscountCalc	100,0 %	8	0	8
main(String[])	100,0 %	1	0	1
calcTax(int, double, int)	100,0 %	1	0	1
calcNoDiscBilR(int)	100,0 %	1	0	1
calcNoDiscBilNR(int)	100,0 %	1	0	1
calcDiscBilR(int)	100,0 %	1	0	1
calcDiscBilNR2(int)	100,0 %	1	0	1
calcDiscBilNR1(int)	100,0 %	1	0	1

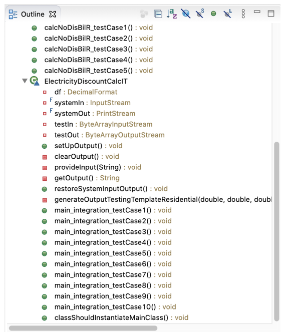
(b) Method coverage

There is still one line (line 19 on Figure 11) that is underline in yellow by Eclipse. This branch is not 100% covered by our tests. In fact some path are not tested by our tests concerning this branch. The solution might be to use Control Structure Testing which is a white-box testing technique where test cases are generated based on the coverage of instructions of code. Of course we should avoid duplicating test cases by adding this technic.

4.2 Coverage optimisation

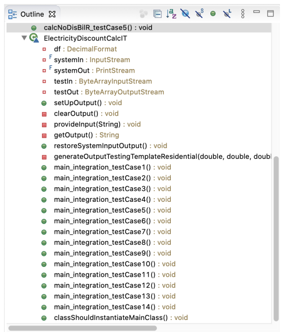
Like mentioned in Section 3.1 we would like to know if the optimisation made in the decision table can have any impact on the coverage percentage. At first we could say yes. We expect the coverage to be higher.

Let's now implement the new test cases



```
Outline
calcNoDisBilR_testCase1() : void
calcNoDisBilR_testCase2() : void
calcNoDisBilR_testCase3() : void
calcNoDisBilR_testCase4() : void
calcNoDisBilR_testCase5() : void
ElectricityDiscountCalcIT
  df : DecimalFormat
  systemIn : InputStream
  systemOut : PrintStream
  testIn : ByteArrayInputStream
  testOut : ByteArrayOutputStream
  setUpOutput() : void
  clearOutput() : void
  provideInput(String) : void
  getOutput() : String
  restoreSystemInputOutput() : void
  generateOutputTestingTemplateResidential(double, double, doubl
  main_integration_testCase1() : void
  main_integration_testCase2() : void
  main_integration_testCase3() : void
  main_integration_testCase4() : void
  main_integration_testCase5() : void
  main_integration_testCase6() : void
  main_integration_testCase7() : void
  main_integration_testCase8() : void
  main_integration_testCase9() : void
  main_integration_testCase10() : void
  classShouldInstantiateMainClass() : void
```

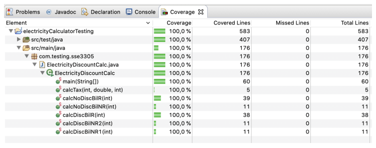
(a) Original test script



```
Outline
calcNoDisBilR_testCase5() : void
ElectricityDiscountCalcIT
  df : DecimalFormat
  systemIn : InputStream
  systemOut : PrintStream
  testIn : ByteArrayInputStream
  testOut : ByteArrayOutputStream
  setUpOutput() : void
  clearOutput() : void
  provideInput(String) : void
  getOutput() : String
  restoreSystemInputOutput() : void
  generateOutputTestingTemplateResidential(double, double, doubl
  main_integration_testCase1() : void
  main_integration_testCase2() : void
  main_integration_testCase3() : void
  main_integration_testCase4() : void
  main_integration_testCase5() : void
  main_integration_testCase6() : void
  main_integration_testCase7() : void
  main_integration_testCase8() : void
  main_integration_testCase9() : void
  main_integration_testCase10() : void
  main_integration_testCase11() : void
  main_integration_testCase12() : void
  main_integration_testCase13() : void
  main_integration_testCase14() : void
  classShouldInstantiateMainClass() : void
```

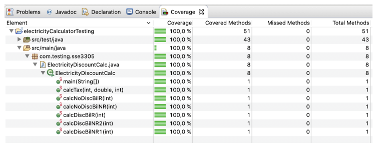
(b) Modified test script

Now we can do a coverage report on eclipse



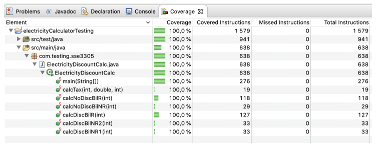
Element	Coverage	Covered Lines	Missed Lines	Total Lines
electricityCalculatorTesting	100,0 %	583	0	583
src/test/java	100,0 %	407	0	407
src/main/java	100,0 %	176	0	176
com.testing.sse3305	100,0 %	176	0	176
ElectricityDiscountCalc.java	100,0 %	176	0	176
ElectricityDiscountCalc	100,0 %	176	0	176
main(String[])	100,0 %	60	0	60
calcTax(int, double, int)	100,0 %	5	0	5
calcNoDiscBilR(int)	100,0 %	39	0	39
calcNoDiscBilNR(int)	100,0 %	11	0	11
calcDiscBilR(int)	100,0 %	38	0	38
calcDiscBilNR2(int)	100,0 %	11	0	11
calcDiscBilNR1(int)	100,0 %	11	0	11

(a) Line coverage



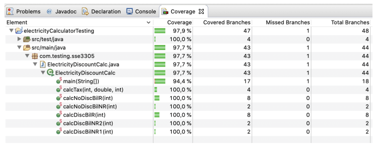
Element	Coverage	Covered Methods	Missed Methods	Total Methods
electricityCalculatorTesting	100,0 %	51	0	51
src/test/java	100,0 %	43	0	43
src/main/java	100,0 %	8	0	8
com.testing.sse3305	100,0 %	8	0	8
ElectricityDiscountCalc.java	100,0 %	8	0	8
ElectricityDiscountCalc	100,0 %	8	0	8
main(String[])	100,0 %	1	0	1
calcTax(int, double, int)	100,0 %	1	0	1
calcNoDiscBilR(int)	100,0 %	1	0	1
calcNoDiscBilNR(int)	100,0 %	1	0	1
calcDiscBilR(int)	100,0 %	1	0	1
calcDiscBilNR2(int)	100,0 %	1	0	1
calcDiscBilNR1(int)	100,0 %	1	0	1

(b) Method coverage



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
electricityCalculatorTesting	100,0 %	1 579	0	1 579
src/test/java	100,0 %	941	0	941
src/main/java	100,0 %	638	0	638
com.testing.sse3305	100,0 %	638	0	638
ElectricityDiscountCalc.java	100,0 %	638	0	638
ElectricityDiscountCalc	100,0 %	638	0	638
main(String[])	100,0 %	276	0	276
calcTax(int, double, int)	100,0 %	19	0	19
calcNoDiscBilR(int)	100,0 %	118	0	118
calcNoDiscBilNR(int)	100,0 %	29	0	29
calcDiscBilR(int)	100,0 %	127	0	127
calcDiscBilNR2(int)	100,0 %	33	0	33
calcDiscBilNR1(int)	100,0 %	33	0	33

(c) Instruction coverage



Element	Coverage	Covered Branches	Missed Branches	Total Branches
electricityCalculatorTesting	97,9 %	47	1	48
src/test/java	100,0 %	4	0	4
src/main/java	97,7 %	43	1	44
com.testing.sse3305	97,7 %	43	1	44
ElectricityDiscountCalc.java	97,7 %	43	1	44
ElectricityDiscountCalc	97,7 %	43	1	44
main(String[])	94,4 %	17	1	18
calcTax(int, double, int)	100,0 %	4	0	4
calcNoDiscBilR(int)	100,0 %	8	0	8
calcNoDiscBilNR(int)	100,0 %	2	0	2
calcDiscBilR(int)	100,0 %	8	0	8
calcDiscBilNR2(int)	100,0 %	2	0	2
calcDiscBilNR1(int)	100,0 %	2	0	2

(d) Branch coverage

Figure 15: Test coverage report with optimisation from Jacoco Eclipse

As you can see on Figure 15, except for the branch coverage, we succesfully increased the coverage percentage to 100%.

Note: see ***integration_test_script_full.pdf*** for more information

5 Conclusion

We saw in this report my strategy to test an electricity discount calculator software.

First of all, in terms of strategy and techniques, I can justify that the suggested solution is the most effective in testing: Basic Path Testing has been really efficient in term of coverage to perform the unit tests. Therefore, we are sure that any functions in the software can be used individually.

Then, thanks to the decision table and the Bundary Value Analysis, the integration tests has covered the main part of the program. Even if there is still one branch that has not been covered by the tests, we can still guarranty the effectiveness of the software.

In addition, the coverage report justify and support the testing solution: we can consider a coverage very successfull hover 99% which is the case here.

Finally, Concerning the readiness of the system for deployment, we can assure that the software is ready to use and can be deployed to the users without having doubts on it performances.

The bottom line is that for further improvements, we are sure that the current code will not be affected by the new changes with this testing solution.