

CSE 565 SVVT Assignment 4: Decision Code Coverage

Siddharth Gianchandani

Part-1

I. Description of tool used

EclEmma is the testing tool used for this assignment, which is based on the JaCoCo code coverage library. This is a free tool that can be used by everyone under the Eclipse Public License for ensuring code coverage. The purpose of this tool is to make sure that the entire code has been tested. For example, with legacy testing methods, you cannot ensure a high level of code coverage from developing test cases, which would create problems in the future but with EclEmma the code segments that have been tested are highlighted and the percentage of code coverage is shown. This feature is also helpful for finding out what parts of the code have not been tested and thus it helps to improve the test cases (referred from [1]).

II. Types of Code Coverage

- a. Instruction or Statement coverage: EclEmma shows the number of instructions that have been executed. This is useful for ensuring that all the sets of instructions have been covered by the test cases.
- b. Line coverage: EclEmma counts the number of lines of code that are executed. This is used to guarantee that all test cases cover all the lines of code.
- c. Branch coverage: EclEmma counts the number of branches that are covered by the test cases. This is useful for ensuring that all the branches in the program have been covered and none of them result in unwanted behaviour.
- d. Method or Functional coverage: EclEmma counts the number of methods that are covered by the test cases. This helps ensure that potential failure and boundary cases have also been covered by the test cases.
- e. Type coverage: EclEmma counts the types that are covered by the test cases. This helps to ensure that the program is running smoothly with the correct types of input.
- f. Complexity or Decision coverage: EclEmma analyses the complexity covered by the test cases. This is useful for ensuring that all the complexity of the program is covered by the test cases.

III. Description of test cases, their scope and coverage

I have developed the following test cases:

1. Test case candy1 is testing for what happens when we give the exact cost of candy.
2. Test case coke1 is testing for what happens when we give the exact cost of coke.
3. Test case coffee1 is testing for what happens when we give the exact cost of coffee.

4. Test case coffee2 is testing what happens when we try to order a coffee with only 40 cents.
5. Test case coke2 is testing for what happens when we try to buy a coke with 24 cents.
6. Test case candy2 is testing for what happens when we try to purchase a candy with 30 cents.
7. Test case candy3 is testing for what happens when we try to purchase a candy with 15 cents.
8. Test case notSameInstances is making sure that no instances of VendingMachine are the same.

The coverage of the test cases for different categories are:

- a. Instruction or statement coverage: 100% (missed 0 out of 99)
- b. Line coverage: 100% (missed 0 out of 26)
- c. Method or functional coverage: 100% (missed 0 out of 2)
- d. Branch coverage: 93.8% (missed 1 out of 16)
- e. Type coverage: 100% (missed 0 out of 1)
- f. Complexity or decision coverage: 90% (missed 1 out of 10)

IV. Screenshot of tool's report showing coverage achieved

Screenshot of instruction (statement) coverage:

/endingMachineTest (Mar 26, 2023 7:25:22 PM)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
SVVT Assignment 1	72.9 %	156	58	214
src	72.9 %	156	58	214
(default package)	72.9 %	156	58	214
StaticAnalysis.java	0.0 %	0	58	58
VendingMachine.java	100.0 %	99	0	99
VendingMachineTest.java	100.0 %	57	0	57

Screenshot for branch coverage:

VendingMachineTest (Mar 26, 2023 7:25:22 PM)

Element	Coverage	Covered Branches	Missed Branches	Total Branches
SVVT Assignment 1	75.0 %	15	5	20
src	75.0 %	15	5	20
(default package)	75.0 %	15	5	20
StaticAnalysis.java	0.0 %	0	4	4
VendingMachine.java	93.8 %	15	1	16
VendingMachineTest.java		0	0	0







Screenshot for line coverage:

VendingMachineTest (Mar 26, 2023 7:25:22 PM)

Element	Coverage	Covered Lines	Missed Lines	Total Lines
SVVT Assignment 1	71.4 %	45	18	63
src	71.4 %	45	18	63
(default package)	71.4 %	45	18	63
StaticAnalysis.java	0.0 %	0	18	18
VendingMachine.java	100.0 %	26	0	26
VendingMachineTest.java	100.0 %	19	0	19







Screenshot for method (functional) coverage:

VendingMachineTest (Mar 26, 2023 7:25:22 PM)

Element	Coverage	Covered Methods	Missed Methods	Total Methods
SVVT Assignment 1	 78.6 %	11	3	14
src	 78.6 %	11	3	14
(default package)	 78.6 %	11	3	14
> StaticAnalysis.java	 0.0 %	0	3	3
> VendingMachine.java	 100.0 %	2	0	2
> VendingMachineTest.java	 100.0 %	9	0	9







Screenshot for type coverage:

VendingMachineTest (Mar 26, 2023 7:25:22 PM)

Element	Coverage	Covered Types	Missed Types	Total Types
SVVT Assignment 1	 66.7 %	2	1	3
src	 66.7 %	2	1	3
(default package)	 66.7 %	2	1	3
> StaticAnalysis.java	 0.0 %	0	1	1
> VendingMachine.java	 100.0 %	1	0	1
> VendingMachineTest.java	 100.0 %	1	0	1

Screenshot for complexity (decision) coverage:

VendingMachineTest (Mar 26, 2023 7:25:22 PM)

Element	Coverage	Covered Complexity	Missed Complexity	Total Complexity
SVVT Assignment 1	 75.0 %	18	6	24
src	 75.0 %	18	6	24
(default package)	 75.0 %	18	6	24
> StaticAnalysis.java	 0.0 %	0	5	5
> VendingMachine.java	 90.0 %	9	1	10
> VendingMachineTest.java	 100.0 %	9	0	9

Screenshot for test cases covered:

```
5 class VendingMachineTest {
6     @Test
7     public void candy1() {
8         assertEquals(VendingMachine.dispenseItem(20, "candy"), "Item dispensed.");
9     }
10
11     @Test
12     public void coffee1() {
13         assertEquals(VendingMachine.dispenseItem(45, "coffee"), "Item dispensed.");
14     }
15
16     @Test
17     public void coke1() {
18         assertEquals(VendingMachine.dispenseItem(25, "coke"), "Item dispensed.");
19     }
20
21     @Test
22     public void coffee2() {
23         assertEquals(VendingMachine.dispenseItem(40, "coffee"), "Item not dispensed, missing 5 cents. Can purchase candy or coke.");
24     }
25
26     @Test
27     public void coke2() {
28         assertEquals(VendingMachine.dispenseItem(24, "coke"), "Item not dispensed, missing 1 cents. Can purchase candy.");
29     }
30
31     @Test
32     public void candy2() {
33         assertEquals(VendingMachine.dispenseItem(30, "candy"), "Item dispensed and change of 10 returned");
34     }
35
36     @Test
37     public void candy3() {
38         assertEquals(VendingMachine.dispenseItem(15, "candy"), "Item not dispensed, missing 5 cents. Cannot purchase item.");
39     }
40
41     @Test
42     public void notSameInstances() {
43         final VendingMachine v1 = new VendingMachine();
44         final VendingMachine v2 = new VendingMachine();
45         assertNotEquals(v1, v2);
46     }
47 }
```

Screenshot for code coverage:

```

public class VendingMachine
{
    public static String dispenseItem(int input, String item)
    {
        int cost = 0;
        int change = 0;
        String returnValue = "";
        if (item == "candy")
            cost = 20;
        if (item == "coke")
            cost = 25;
        if (item == "coffee")
            cost = 45;

        if (input > cost)
        {
            change = input - cost;
            returnValue = "Item dispensed and change of " + Integer.toString(change) + " returned";
        }
        else if (input == cost)
        {
            change = 0;
            returnValue = "Item dispensed.";
        }
        else
        {
            change = cost - input;
            if(input < 45)
                returnValue = "Item not dispensed, missing " + Integer.toString(change) + " cents. Can purchase candy or coke.";
            if(input < 25)
                returnValue = "Item not dispensed, missing " + Integer.toString(change) + " cents. Can purchase candy.";
            if(input < 20)
                returnValue = "Item not dispensed, missing " + Integer.toString(change) + " cents. Cannot purchase item.";
        }

        return returnValue;
    }
}

```

V. Evaluation of tool's usefulness

The tool, EclEmma, is very useful in helping developers and testers with keeping tracks of the sections of the code that are covered by test cases and gives accurate information about the code coverage. EclEmma is concise and precise when it comes to code coverage. It also does not require any initial setup and this helps prevent versioning issues with the project as a whole. It also helps automate testing techniques like JUnit unit tests which often use code coverage analysis. Another advantage of EclEmma is that it maintains a record of the parts of the code that are executed when a certain program is launched. All these features result in less time needed for testing, less issues with the code and a reduced risk of defects in the code.

Part-2

I. Description of Data Flow Anomalies

Data anomalies are found when doing a static test or a box test. We can find two characters to represent the data flow anomalies based on the order of our actions. They are used (U), killed (K) and defined (D). The nine possible combinations are: DD, KK, UU, DK, DU, KD, KU, UD, and UK (referred from [2]).

The 2 data anomalies detected for “cost” and “output” are of the DD type as shown in screenshot 5. The reason is that these variables are defined at lines 21 and 22 respectively and are redefined as the program executes based on certain conditions. This results in the variables being defined twice, which is considered as a data anomaly.

II. Screenshot of analysis performed

Screenshot 1:

The screenshot shows an IDE with a Java file named `StaticAnalysis.java`. The code defines a `StaticAnalysis` class with a `main` method and a `calculateCost` method. The `main` method calls `calculateCost` with arguments `5, 10, "Electronics"` and `2, 3, "Clothing"`. The `calculateCost` method calculates the cost based on the product type and returns a string. The violations outline table shows the following data:

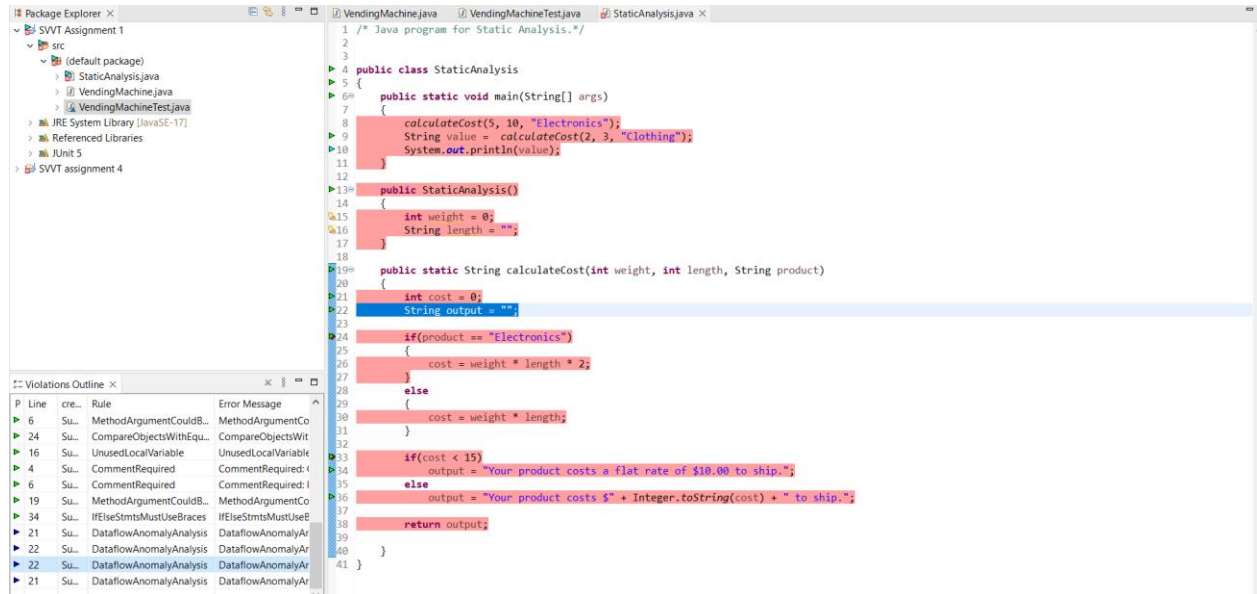
Line	cre...	Rule	Error Message
6	Su...	MethodArgumentCouldB...	MethodArgumentCo
24	Su...	CompareObjectsWithEqu...	CompareObjectsWit
16	Su...	UnusedLocalVariable	UnusedLocalVariable
4	Su...	CommentRequired	CommentRequired: I
6	Su...	CommentRequired	CommentRequired: I
19	Su...	MethodArgumentCouldB...	MethodArgumentCo
34	Su...	IFElseStmtsMustUseBraces	IFElseStmtsMustUseB
21	Su...	DataflowAnomalyAnalysis	DataflowAnomalyAr
22	Su...	DataflowAnomalyAnalysis	DataflowAnomalyAr
22	Su...	DataflowAnomalyAnalysis	DataflowAnomalyAr
21	Su...	DataflowAnomalyAnalysis	DataflowAnomalyAr

Screenshot 2:

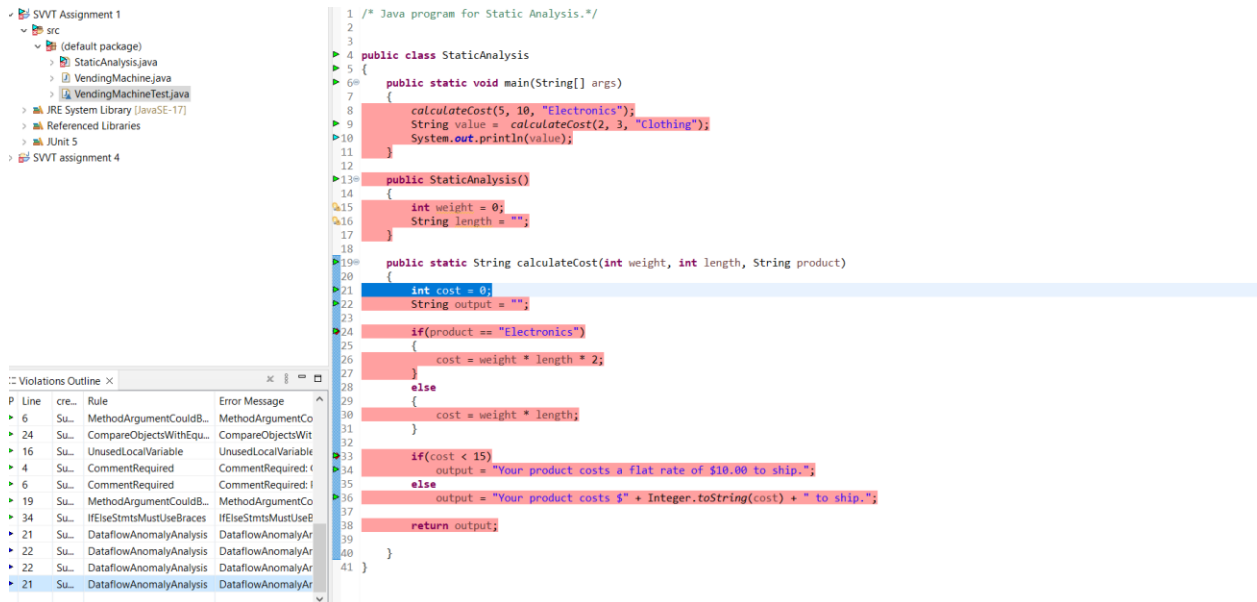
The screenshot shows the same IDE as Screenshot 1, but with a different set of violations. The violations outline table shows the following data:

P Line	cre...	Rule	Error Message
6	Su...	MethodArgumentCouldB...	MethodArgumentCo
24	Su...	CompareObjectsWithEqu...	CompareObjectsWit
16	Su...	UnusedLocalVariable	UnusedLocalVariable
4	Su...	CommentRequired	CommentRequired: I
6	Su...	CommentRequired	CommentRequired: I
19	Su...	MethodArgumentCouldB...	MethodArgumentCo
34	Su...	IFElseStmtsMustUseBraces	IFElseStmtsMustUseB
21	Su...	DataflowAnomalyAnalysis	DataflowAnomalyAr
22	Su...	DataflowAnomalyAnalysis	DataflowAnomalyAr
21	Su...	DataflowAnomalyAnalysis	DataflowAnomalyAr

Screenshot 3:



Screenshot 4:



Screenshot 5 (showing complete messages from the PMD tool):

21	Su...	DataflowAnomalyAnalysis	DataflowAnomalyAnalysis: Found 'DD'-anomaly for variable 'cost' (lines '21'-'26').
22	Su...	DataflowAnomalyAnalysis	DataflowAnomalyAnalysis: Found 'DD'-anomaly for variable 'output' (lines '22'-'34').
22	Su...	DataflowAnomalyAnalysis	DataflowAnomalyAnalysis: Found 'DD'-anomaly for variable 'output' (lines '22'-'36').
21	Su...	DataflowAnomalyAnalysis	DataflowAnomalyAnalysis: Found 'DD'-anomaly for variable 'cost' (lines '21'-'30').

III. Evaluation of tool's usefulness

PMD is a popular static source code analyzer for Eclipse and IntelliJ. PMD is mainly concerned with Java and Apex but supports 14 languages. It finds common programming flaws like unused variables, empty catch blocks and unnecessary object creation through various built-in checks. It also gives everyone access to a robust API so that people can create their own rules in Java. It is one of the best integration tools that also imposes coding standards. This tool works best when included in the building process of the codebase so that it can be utilized as a quality control tool to establish a coding standard (referred from [3]).

References:

- [1] <https://www.eclemma.org/>
- [2] https://www.tutorialspoint.com/software_testing_dictionary/anomaly.htm
- [3] <https://docs.pmd-code.org/latest/>