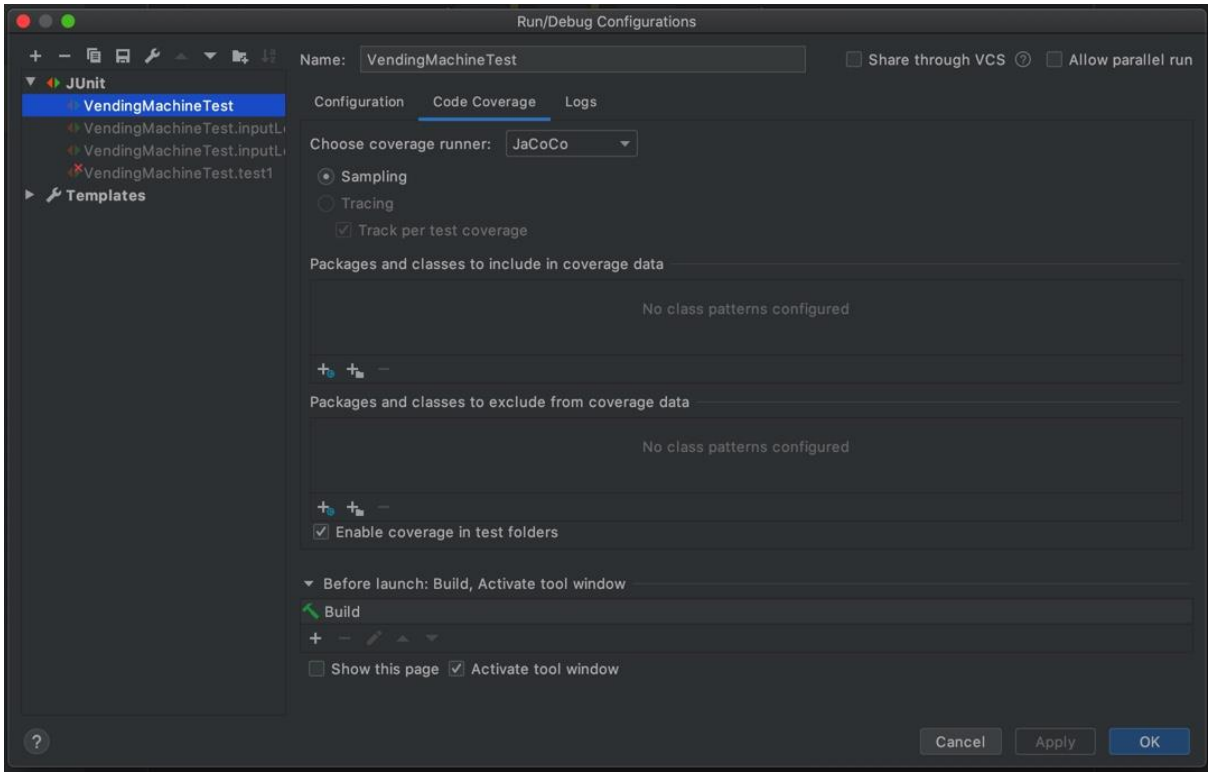
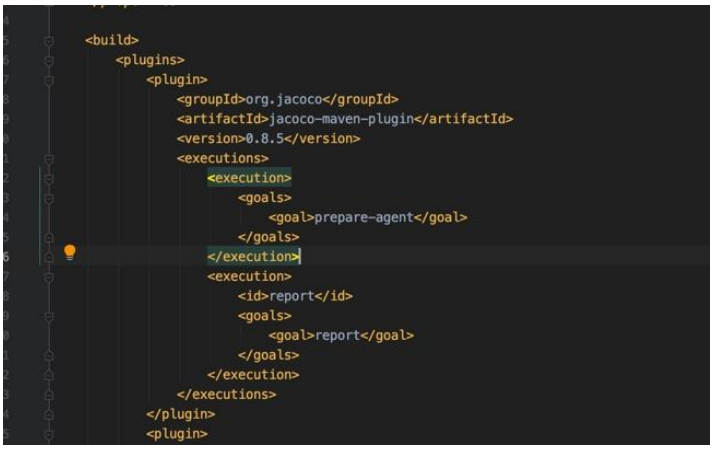
Part 1 – Jacoco : statement & decision coverage

1. **Description** : Jacoco (Java Code Coverage) is a free code coverage library for Java, that's also a report generator.
   IntelliJ IDEA has a built-in Jacoco plugin for code coverage, in order to use it, we just have to select it in our Tests configuration :



I've also used jacoco-maven-plugin in project's pom file (maven build tool), in order to let the jacoco run and generate a report everytime the build runs.



2. Test cases

No	input	item	Expected output
1	30	candy	"Item dispensed and change of 10 returned"
2	25	coke	"Item dispensed."
3	35	coffee	"Item not dispensed, missing 10 cents. Can purchase candy or coke."
4	20	coke	"Item not dispensed, missing 5 cents. Can purchase candy."
5	15	candy	"Item not dispensed, missing 5 cents. Cannot purchase item."

In order to use the jacoco code coverage tool we have to write a test suite (junit tests) and let them run with coverage, here is the java test class with all the above test cases :

VendingMachineTest.java

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class VendingMachineTest {

    private VendingMachine vendingMachine;

    @BeforeEach
    public void setup(){
        vendingMachine = new VendingMachine();
    }

    @Test
    public void inputBiggerCostAndChooseCandy() {
        String returnValue = vendingMachine.dispenseItem( input: 30,  item: "candy");
        assertEquals(returnValue,  actual: "Item dispensed and change of 10 returned");
    }

    @Test
    public void inputEqualsCostAndChooseCoke() {
        String returnValue = vendingMachine.dispenseItem( input: 25,  item: "coke");
        assertEquals(returnValue,  actual: "Item dispensed.");
    }

    @Test
    public void inputLowerCostAndChooseCoffee() {
        String returnValue = vendingMachine.dispenseItem( input: 35,  item: "coffee");
        assertEquals(returnValue,  actual: "Item not dispensed, missing 10 cents. Can purchase candy or coke.");
    }

    @Test
    public void inputLowerCostAndChooseCoke() {
        String returnValue = vendingMachine.dispenseItem( input: 20,  item: "coke");
        assertEquals(returnValue,  actual: "Item not dispensed, missing 5 cents. Can purchase candy.");
    }

    @Test
    public void inputLowerCostAndChooseCandy() {
        String returnValue = vendingMachine.dispenseItem( input: 15,  item: "candy");
        assertEquals(returnValue,  actual: "Item not dispensed, missing 5 cents. Cannot purchase item.");
    }

}
```

3. Achieved coverage

Opening the report.html generated by jacoco results in the following :

cse565_Project2

cse565_Project2

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
default		100 %		93 %	1	10	0	24	0	2	0	1
Total	0 of 71	100 %	1 of 16	93 %	1	10	0	24	0	2	0	1

cse565_Project2 > default

default

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
VendingMachine		100 %		93 %	1	10	0	24	0	2	0	1
Total	0 of 71	100 %	1 of 16	93 %	1	10	0	24	0	2	0	1

cse565_Project2 > default > VendingMachine

VendingMachine

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
dispenseItem(int, String)		100 %		93 %	1	9	0	23	0	1
VendingMachine()		100 %		n/a	0	1	0	1	0	1
Total	0 of 71	100 %	1 of 16	93 %	1	10	0	24	0	2

cse565_Project2 > default > VendingMachine.java

VendingMachine.java

```
1.  /* Java program for Vending Machine. The class takes in the 2 parameters
2.  and returns whether the item can be dispensed or not */
3.
4.  public class VendingMachine
5.  {
6.
7.      public static String dispenseItem(int input, String item)
8.      {
9.          int cost = 0;
10.         int change = 0;
11.         String returnValue = "";
12.     ◆   if (item == "candy")
13.             cost = 20;
14.     ◆   if (item == "coke")
15.             cost = 25;
16.     ◆   if (item == "coffee")
17.             cost = 45;
18.
19.     ◆   if (input > cost)
20.         {
21.             change = input - cost;
22.             returnValue = "Item dispensed and change of " + Integer.toString(change) + " returned";
23.         }
24.     ◆   else if (input == cost)
25.         {
26.             change = 0;
27.             returnValue = "Item dispensed.";
28.         }
29.         else
30.         {
31.             change = cost - input;
32.         ◆   if(input < 45)
33.                 returnValue = "Item not dispensed, missing " + Integer.toString(change) + " cents. Can purchase candy or coke.";
34.     ◆       if(input < 25)
35.                 returnValue = "Item not dispensed, missing " + Integer.toString(change) + " cents. Can purchase candy.";
36.     ◆       if(input < 20)
37.                 returnValue = "Item not dispensed, missing " + Integer.toString(change) + " cents. Cannot purchase item.";
38.         }
39.
40.         return returnValue;
41.
42.     }
43. }
```

4. Evaluation of tool

From the above report we can see the statement coverage (missed instructions 0 of 71, cov 100%) has been fully achieved.

The tool is also showing 1 of 16 branches hasn't been covered yet, because we have 8 IF statements, and for every one it's expecting tests that evaluate those branches to TRUE and FALSE. However we have one branch that can't be evaluated to false and that's the highlighted branch in the above example. Every test case with input greater than 45 would evaluate one of the above IF-branches and never reach the final ELSE.
Jacoco's report is correctly showing that 1 of 16 branches is lacking coverage and calculating the total percentage of decision (branch) coverage to be 93%.

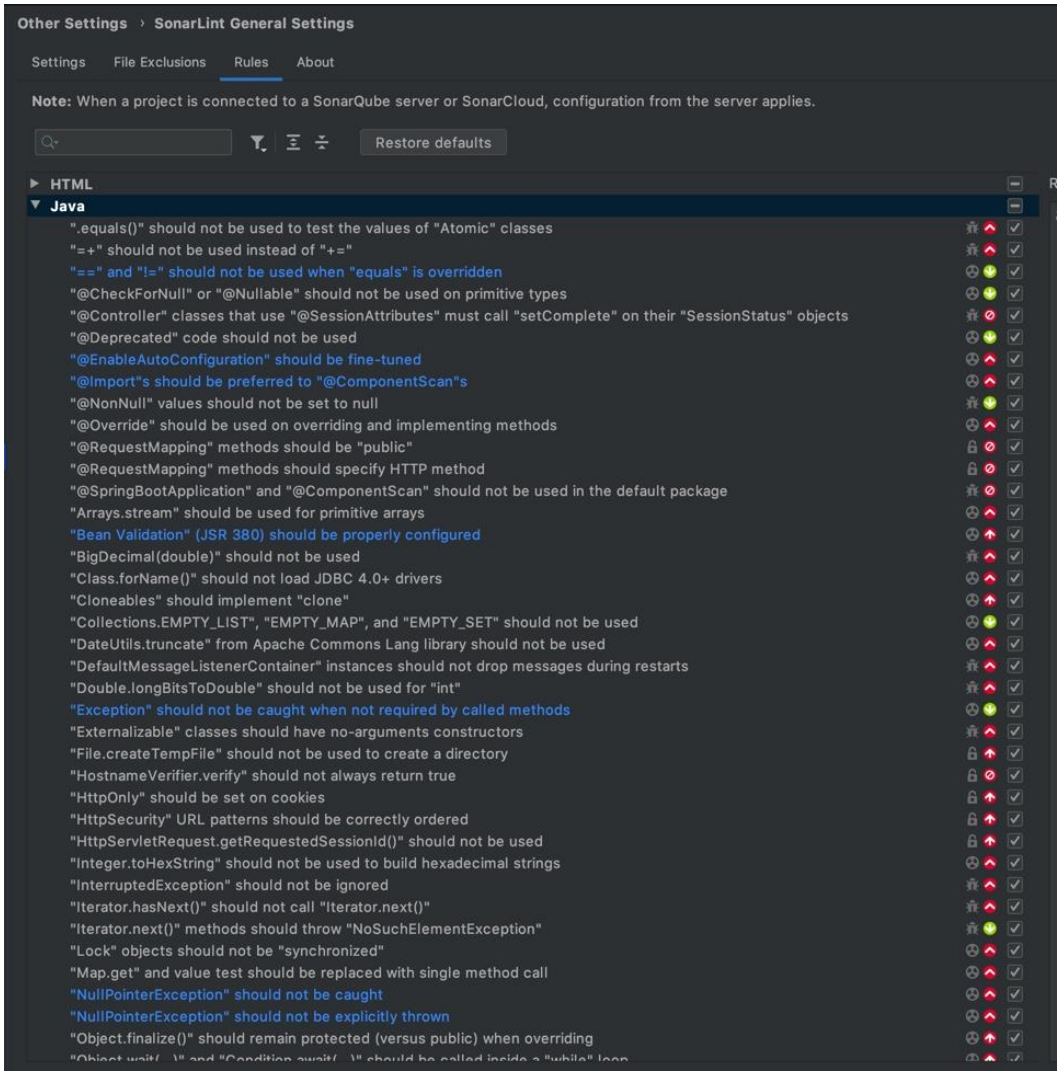
Clicking on the VendingMachine.java class in the report.html, opens up another view, where the coverage (statement & decision) is displayed for each method of the class, including the constructor, this way we know which exact method of the class is lacking code coverage, one interesting thing to note is that test cases have to also cover invoking the constructor of the class, even if the tested method is static ☺

The generated report, with all the information about statement and decision coverage, for every class and every method of our project (imagine having to do 100% coverage for 1000 classes!) makes this a really useful and powerful tool, that can show us graphically which statement or branch hasn't been covered by tests, so we can improve our coverage faster.

Part 2 – SonarLint : static analysis tool

1. Description

SonarLint is an IDE plugin/extension that provides static analysis of the code as you write it and helps in detecting and fixing code quality issues. SonarLint has a default list of rules – a dictionary of code quality issues, that can checked/unchecked and are language specific. Each rule has a criticality flag – critical code smell, major code smell, blocker bug, minor bug. In IntelliJ IDEA I've activated all rules for SonarLint>Java and this lead to finding 26 code quality issues in StaticAnalysis.java file.



2. Data flow anomalies

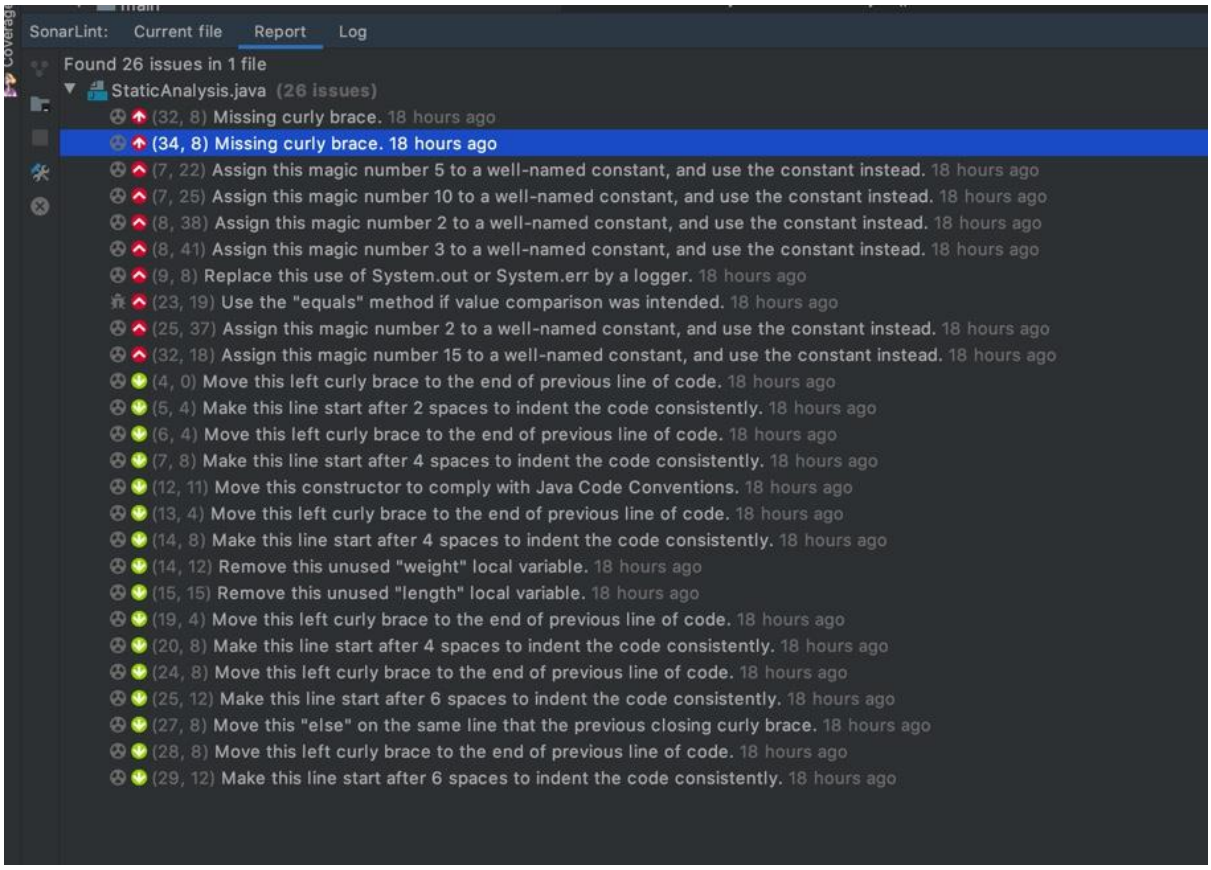
From the list of found issues , there are 2 that are **data flow anomalies**:

- Unused weight local variable, unused length local variable
- Using “==” instead of equals if value comparison was intended line 23

<u>Other found issues were :</u>

- Major Code smells
    - Assign magic numbers like 5,10,2,3, to a well-named constant and use the constant instead
    - Replace the use of System.out or System.err by a logger
    - Missing curly braces

3. Screenshot of SonarLint's report



4. Evaluation of SonarLint

SonarLint is an easy to install and use plugin (you just have to hit Analyse with SonarLint) and can detect serious issues and bugs, but also small formatting and coding/typing errors, that we might have overlooked when coding, it also helps to keep the code clean and arranged, providing suggestions for every found issue.