

Second Project

CSE565 – Software Verification and Validation

Project 2 Analyzing Code Coverage

Part 1

Description of the tool and type of coverage provided

The first part of this project examines statement and decision code coverage while formulating a testing strategy and utilizing a tool. The link below was used to download the Eclipse IDE: <https://www.eclipse.org/downloads/packages/release/luna/sr1/eclipse-ide-java-developers>.

Furthermore, while working within the Eclipse IDE, I utilized the Eclipse Marketplace to obtain EclEmma 3.1.7, which is a freely available Java code coverage tool. According to the documentation provided by EclEmma, the tool provides diverse types of code coverage analysis, including instructions, Lines (representing Statement Coverage), Branches (as referred to as Decision Coverage), Classes as well as Methods. It is worth mentioning that line coverage functions similarly to statement coverage, offering a more comprehensive perspective on code coverage by assessing the execution of entire lines. EclEmma offers a wide range of code coverage analysis options, and its user-friendly interface simplifies the task of performing quick coverage assessments. It accomplishes this through the use of color-coded results, where red denotes no coverage, green indicates full coverage, as well as yellow represents a limited extent.

Moreover, EclEmma includes a feature that enables the export of coverage data, simplifying the examination of data in various structures, such as XML, HTML, CSV, and operational data files.

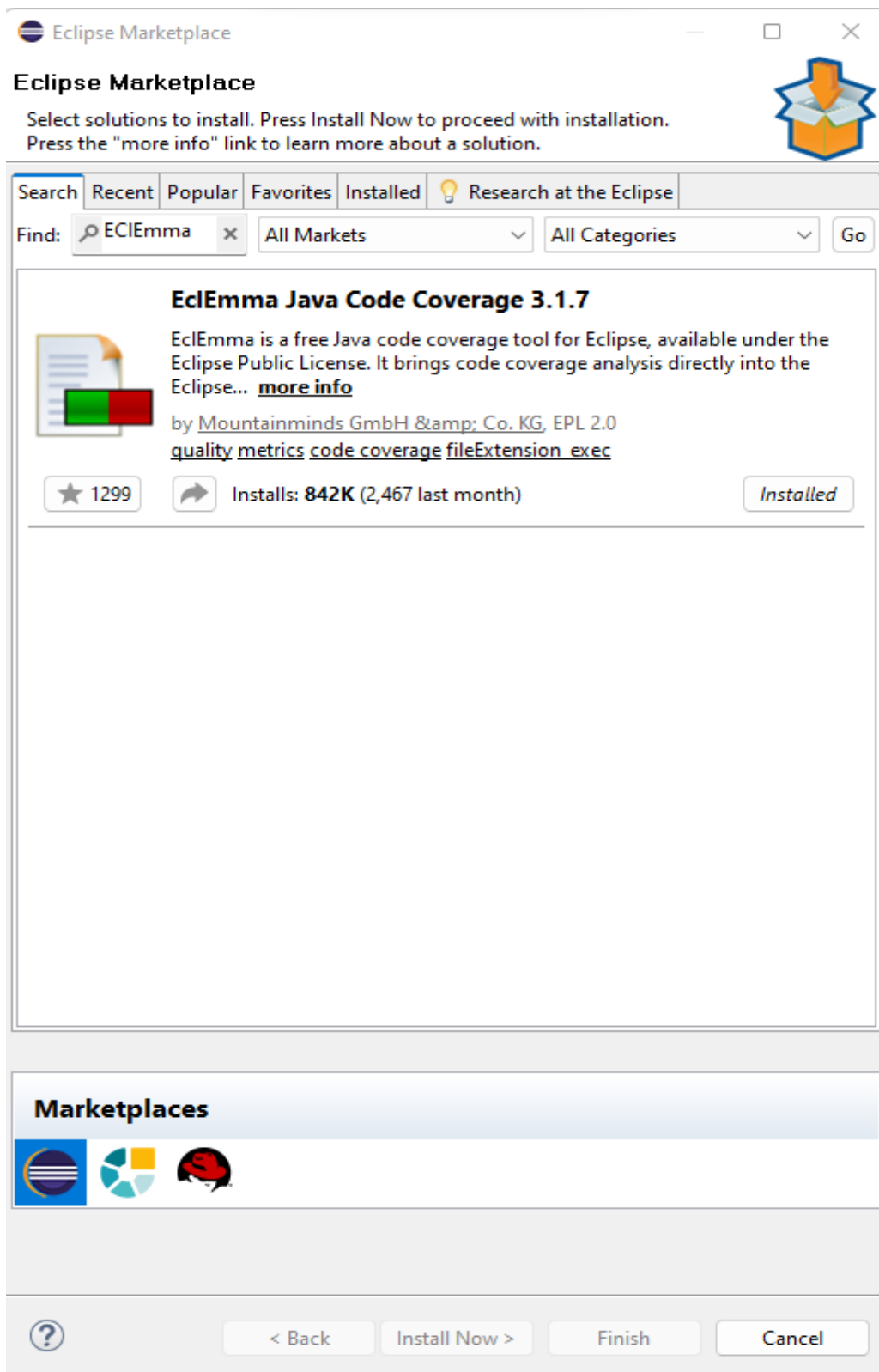


Figure 1: Screenshot of EclEmma installation in Eclipse IDE.

Development and description of a test case set

Below Table 1 displays the collection of test cases employed to attain the targeted level of decision and statement coverage. In each test case, the input provided to the vending machine and the requested item are depicted, showcasing the extent of coverage achieved.

Test No.	Input	Item
1	22	coke
2	25	candy
3	25	coke
4	44	coffee
5	18	candy
6	45	coffee

Table 1. The set of test cases for TestVendingMachine program.

Test case 1:

For input 22 cents and the item coke vending machine should not dispense the coke item because to get coke item input should be 25 cents. So, it's needed 3 more cents to get coke item. Hence, expected output would be an Item dispensed, missing 3 cent. Can purchase candy.

Test case 2:

For input 25 cents and the item candy vending machine should dispense the candy item because to get candy item input should be 20 cents and here it is more than 20. So, the expected output would be an Item dispensed and change of 5 returned.

Test case 3:

Here input is 25 and the item is coke. So, for this input parameter vending machines should dispense coke items as it is the exact price for getting coke according to given vending machine program. Hence, expected output is Item dispensed.

Test case 4:

For test case 4, I put input 44 and item coffee. Here vending machine should not dispense the coffee item, as to get coffee input must be 45 cents but here it is less than 45. So, the expected output would be an Item not dispensed, missing 1 cent. Can purchase candy or coke.

Test case 5:

Here input is 18 and selected item is candy. In this case, vending machines should not dispense the candy item because I should put 20 cents to get the desired item. Therefore, the expected output from the vending machine would be an Item not dispensed, missing 2 cents. Cannot purchase item.

Test case 6:

For input 45 and the selected item coffee vending machine should dispense the coffee item, as for getting coffee input should be 45 due to given vending machine program specification. So, the expected output would be an Item dispensed.

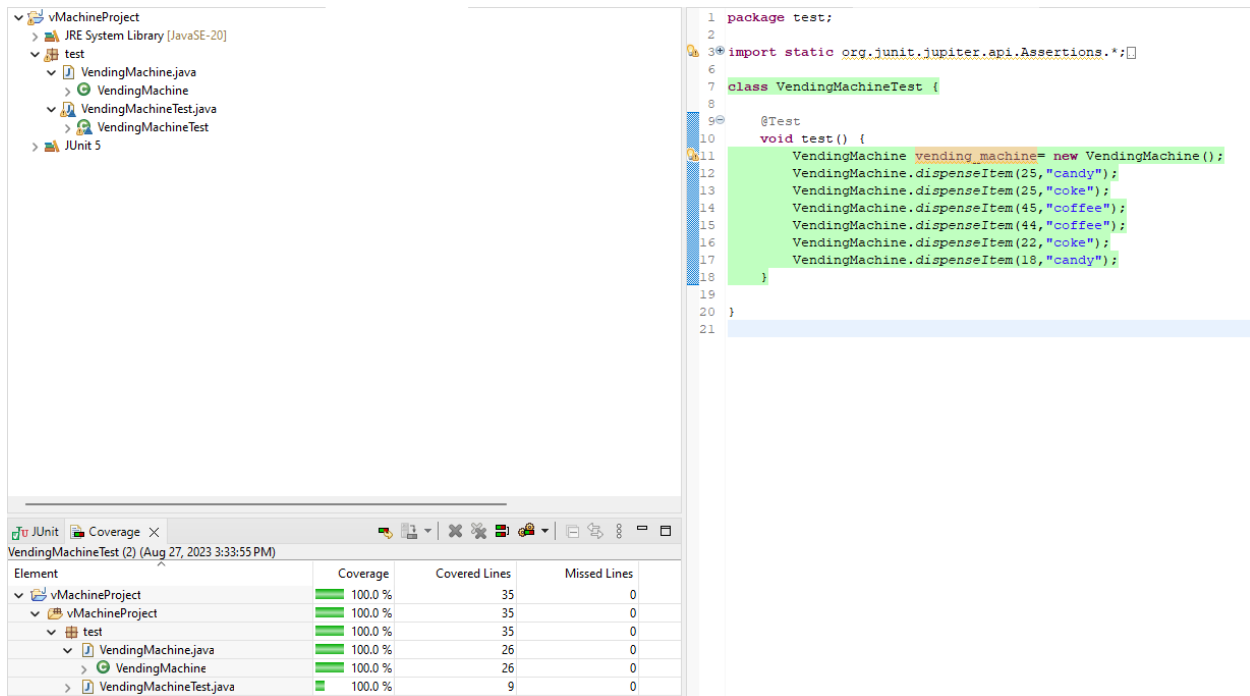


Figure 2. Eclipse IDE screenshot from the test case.

Reporting out and discussion of the code coverage

Figures 3 through 5 depict screen captures of the Test Coverage project within the Eclipse IDE interface, showcasing the results for statement coverage and decision coverage.. Additionally, I have added screenshots of test coverage, for example, method coverage, complexity coverage, line coverage results in figure 6 to figure 8.

```

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

```

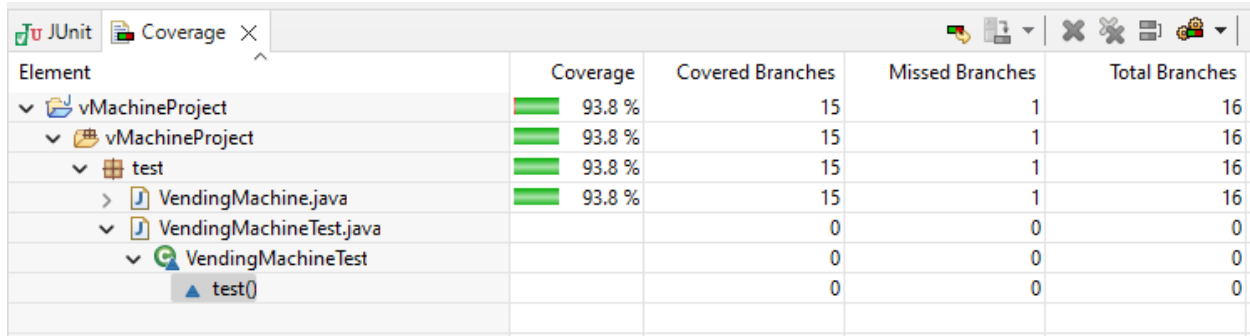
Figure 3: Test Coverage and Eclipse IDE screenshot.

Instruction /Statement Coverage:

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
▼ vMachineProject	100.0 %	131	0	131
▼ vMachineProject	100.0 %	131	0	131
▼ test	100.0 %	131	0	131
> VendingMachine.java	100.0 %	99	0	99
▼ VendingMachineTest.java	100.0 %	32	0	32
▼ VendingMachineTest	100.0 %	32	0	32
▲ test()	100.0 %	29	0	29

Figure 4: Screenshot of 100% Instruction /Statement Coverage.

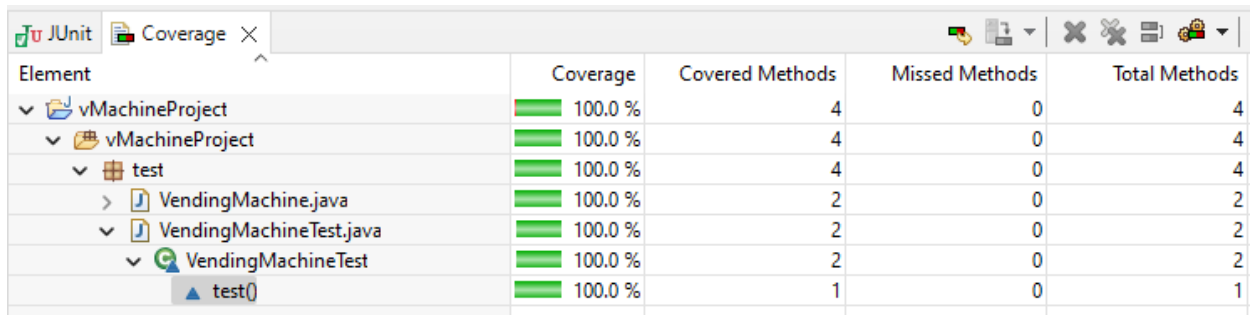
Branch/Decision Coverage:



Element	Coverage	Covered Branches	Missed Branches	Total Branches
vMachineProject	93.8 %	15	1	16
vMachineProject	93.8 %	15	1	16
test	93.8 %	15	1	16
VendingMachine.java	93.8 %	15	1	16
VendingMachineTest.java		0	0	0
VendingMachineTest		0	0	0
test()		0	0	0

Figure 5: Screenshot of 93.8% Branch / Decision Coverage.

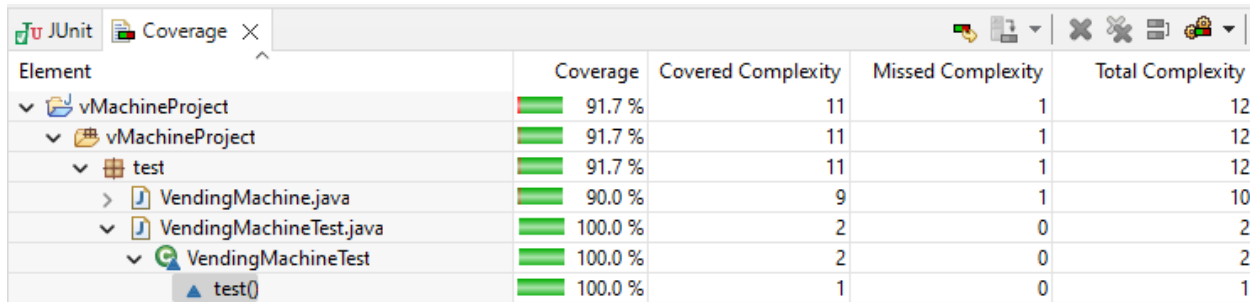
Method Coverage:



Element	Coverage	Covered Methods	Missed Methods	Total Methods
vMachineProject	100.0 %	4	0	4
vMachineProject	100.0 %	4	0	4
test	100.0 %	4	0	4
VendingMachine.java	100.0 %	2	0	2
VendingMachineTest.java	100.0 %	2	0	2
VendingMachineTest	100.0 %	2	0	2
test()	100.0 %	1	0	1

Figure 6: Screenshot of 100 % Method Coverage.

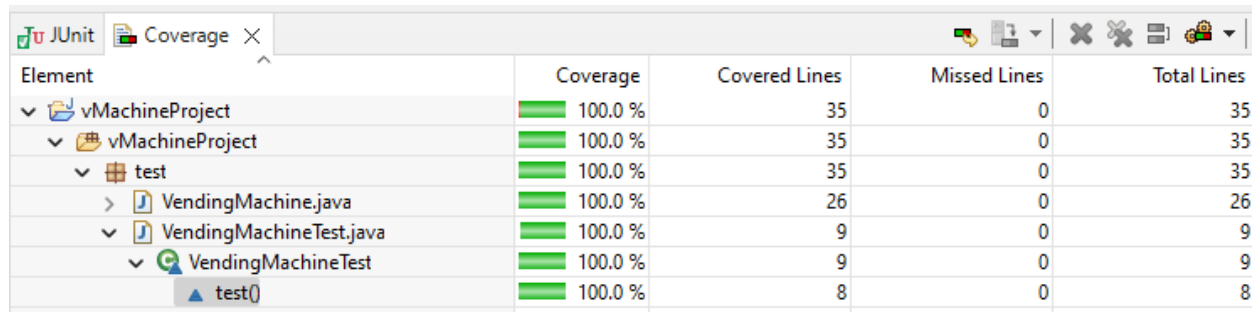
Complexity Coverage:



Element	Coverage	Covered Complexity	Missed Complexity	Total Complexity
vMachineProject	91.7 %	11	1	12
vMachineProject	91.7 %	11	1	12
test	91.7 %	11	1	12
VendingMachine.java	90.0 %	9	1	10
VendingMachineTest.java	100.0 %	2	0	2
VendingMachineTest	100.0 %	2	0	2
test()	100.0 %	1	0	1

Figure 7: Screenshot of Complexity Coverage.

Line Coverage:



Element	Coverage	Covered Lines	Missed Lines	Total Lines
▼ vMachineProject	100.0 %	35	0	35
▼ vMachineProject	100.0 %	35	0	35
▼ test	100.0 %	35	0	35
> VendingMachine.java	100.0 %	26	0	26
▼ VendingMachineTest.java	100.0 %	9	0	9
▼ VendingMachineTest	100.0 %	9	0	9
▲ test()	100.0 %	8	0	8

Figure 8: Screenshot of 100% Line Coverage.

In the provided screenshot (Figure -4), it's evident that we have attained a complete statement coverage of 100% with no missed instructions. However, in Figure -3, the highlighted branch (line-29 in yellow) remains unassessed due to “false” indication, indicating an issue. For any test case where the input value exceeds 45, it will follow one of the mentioned IF-branches and never reach the concluding ELSE statement. Consequently, the decision coverage stands at 93%. The green color indicates adequate code coverage, while the red color signifies sections that have not been executed in any test case.

Part 2

Tool description and anomaly types covered

For the project's second phase, the aim was to choose an appropriate Static Analyzer tool designed to detect potential irregularities in data flow within the Java Programming Language. To accomplish it, I set up Eclipse IDE and subsequently installed Solar Lint via Eclipse Marketplace. Sonar Lint operates as an IDE plugin or extension, conducting static code analysis at the time of the process of development and aiding in the detection and resolution of code quality issues. The software includes preconfigured rules customized for specific programming languages, enabling the tool to efficiently identify and resolve detected coding issues. These rules are classified with labels like critical code smell, major code smell minor bug, and similar designations.

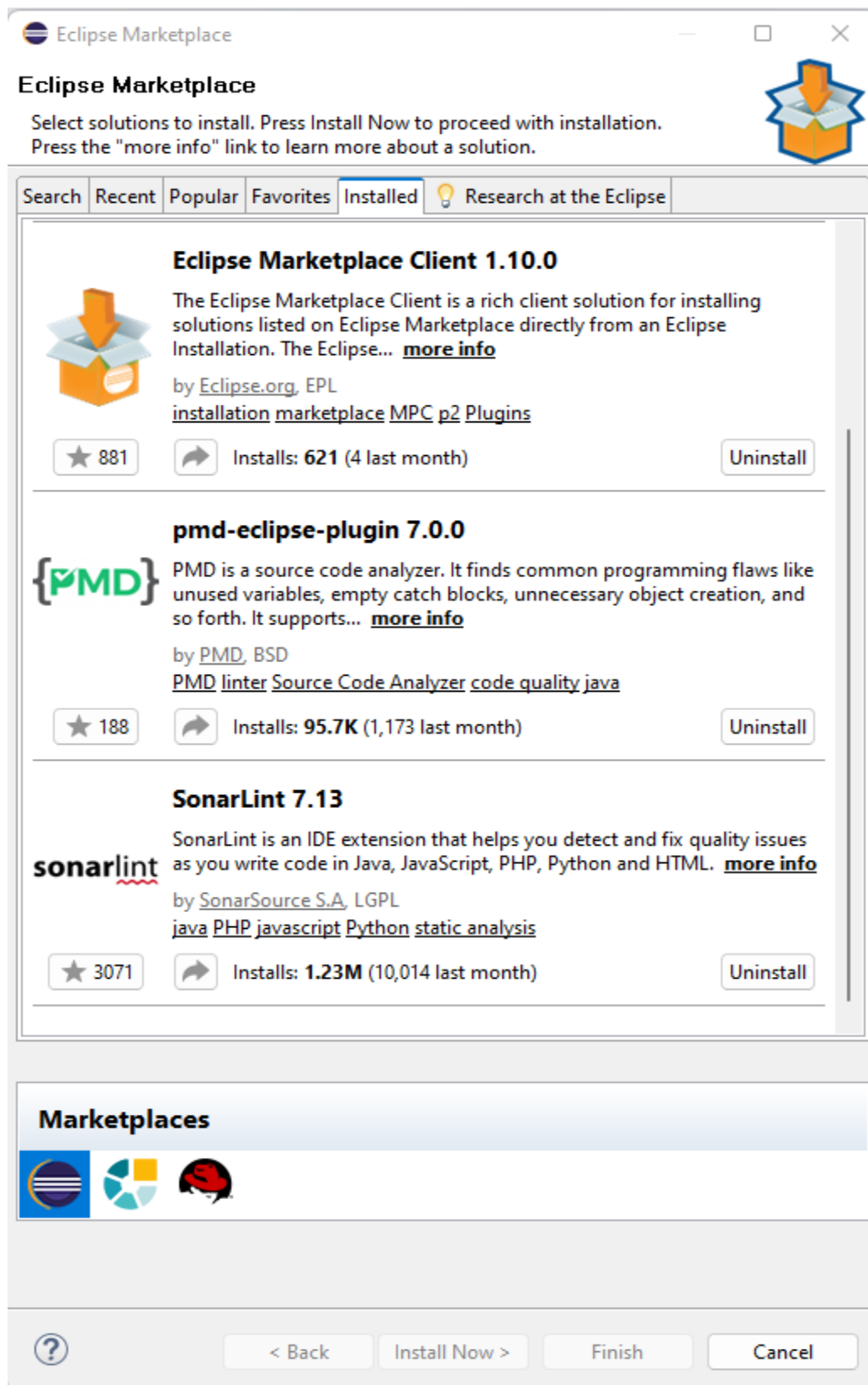


Figure 9: Screenshot of Eclipse IDE with Sonar lint installed.

After establishing the static Analysis project within Eclipse IDE, Sonar Lint conducted a scan of the source code (Static Analysis) to identify potential issues. The tool effectively identified two anomalies, which is elaborated in the subsequent section.

Anomalies detected and described by the tool

Among the identified problems, two of them are categorized as data flow irregularities:

Anomaly 1:

Variable defined but never used. Variables “weight” and “length” are defined in line 13 and 14(Figure 12) of the class but not used in the entire class.

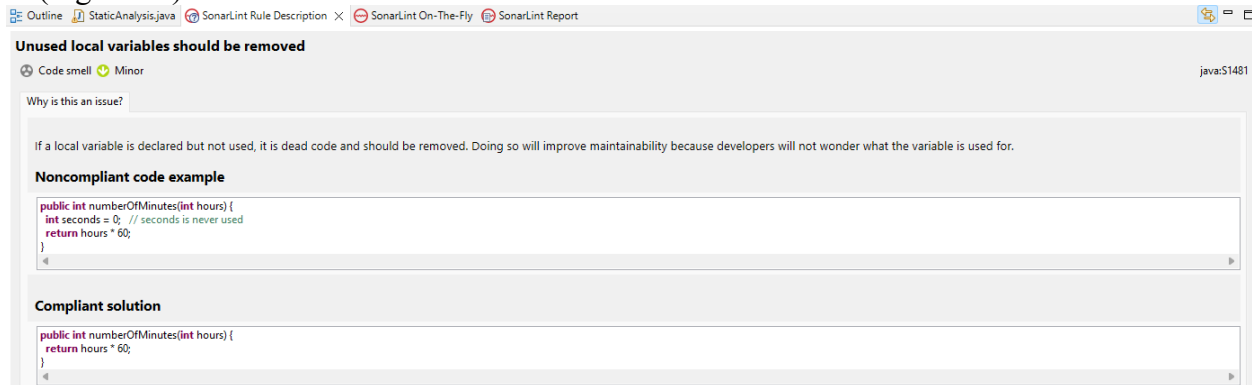


Figure 10: Screenshot of data flow anomaly related to unused local variables.

Anomaly 2:

Java. Lang is compared using this code. Using “==” instead of equals on Line 22 (Figure 12)



Figure 11: Screenshot of data flow anomaly related to Strings and Boxed types.

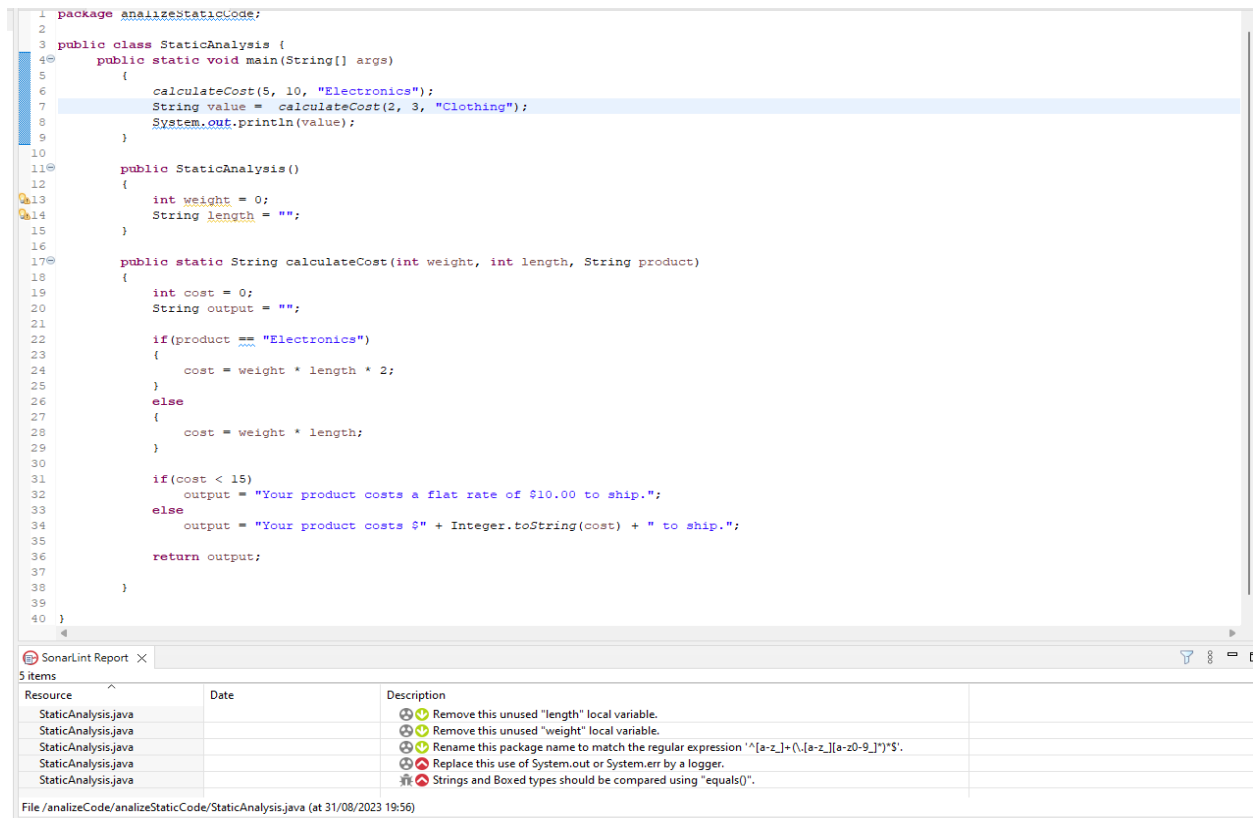


Figure 12: Screenshot of analysis performed with report.

Assessment of the tool

Feature and functionalities:

With support for 25 languages and compatibility with 11 integrated development environments (IDEs), Sonar Lint utilizes a vast repository of over 5000 coding and language-specific rules. These files enable it to promptly identify bugs, vulnerabilities, and code quality issues, while also offering contextual guidance on resolving them.

After installing Sonar Lint within my Eclipse IDE, I imported the “Static Analysis” project. Upon accessing the project, I located and selected the “Sonar Lint” icon. By clicking this icon, I opted for the “Analyze” function, which triggered the automatic generation of a report highlighting multiple anomalies. Subsequently, I right-clicked on a specific error and chose “Rule Description,” which provided insights into the underlying causes of the issue and offered guidance on how to rectify it.

Coverage Provided:

This tool produces a thorough violation report encompassing diverse data flow anomalies. Furthermore, it furnishes in-depth descriptions for each anomaly, directing to the exact code lines responsible for those. This capability greatly simplifies the process of addressing and rectifying issues within the program.

Usability:

Sonar Lint boasts a user-friendly installation and usage experience. The installation is straightforward since Sonar Lint functions as a plugin for the Eclipse IDE, and the Eclipse Marketplace streamlines the setup process. The user interface is intuitive, capable of identifying significant issues and bugs. Moreover, it aids in spotting formatting and typographical errors that might have been overlooked during coding, contributing to maintaining clean and well-structured code.

Reference:

- a. "Eclipse IDE for Java Developers." Eclipse Downloads, The Eclipse Foundation,
<https://www.eclipse.org/downloads/packages/release/luna/sr1/eclipse-ide-java-developers>