

Project 2: Structural-Based Testing

Lucy Zhang

Ira A. Fulton Schools of Engineering, Arizona State University
lzhan370@asu.edu

Introduction

The central aim of this project revolves around the deep exploration and analysis of code coverage utilizing structural-based testing techniques.

The objectives are:

- Analyzing code coverage employing both statement and decision coverage methodologies.
- Crafting a robust set of test cases grounded on given stipulations.
- Gaining insights into the various data flow anomalies.

This undertaking is bifurcated into two critical segments, with each focusing on refining certain proficiencies in software testing and code examination.

Part 1: Code Coverage Analysis for Vending Machine Application

Description of the tool and type of coverage

Tool Used: JaCoCo in Eclipse IDE

JaCoCo is a widely used code coverage library for Java, and it offers a comprehensive set of features to track and measure the coverage of the Java code from unit tests, integration tests, and even manual tests.

Type of Code Coverage: JaCoCo offers an exhaustive analysis of Java code, including aspects such as:

- Statement Coverage: Measures the number of executed statements in comparison to the total.
- Decision Coverage: Assesses the true and false outcomes of decisions.

Development and description of a set of test cases

Test Case Development: Multiple test cases were formulated to validate the code in VendingMachine.java, targeting diverse scenarios, such as exact change, excess amount, insufficient funds, and incorrect input.

```

VendingMachine.java  testvm.java X
1  package test;
2
3  import static org.junit.jupiter.api.Assertions.*;
4
5
6  class testvm {
7
8
9  @Test
10 public void testPurchaseCandyExactChange() {
11     assertEquals("Item dispensed.", VendingMachine.dispenseItem(20, "candy"));
12 }
13
14 @Test
15 public void testPurchaseCokeExactChange() {
16     assertEquals("Item dispensed.", VendingMachine.dispenseItem(25, "coke"));
17 }
18
19 @Test
20 public void testPurchaseCoffeeExactChange() {
21     assertEquals("Item dispensed.", VendingMachine.dispenseItem(45, "coffee"));
22 }
23
24 @Test
25 public void testPurchaseCandyExtraChange() {
26     assertEquals("Item dispensed and change of 10 returned", VendingMachine.dispenseItem(30, "candy"));
27 }
28
29 @Test
30 public void testPurchaseCokeExtraChange() {
31     assertEquals("Item dispensed and change of 5 returned", VendingMachine.dispenseItem(30, "coke"));
32 }
33
34 @Test
35 public void testPurchaseCoffeeExtraChange() {
36     assertEquals("Item dispensed and change of 10 returned", VendingMachine.dispenseItem(55, "coffee"));
37 }
38
39 @Test
40 public void testPurchaseCandyNotEnoughChange() {
41     assertEquals("Item not dispensed, missing 5 cents. Cannot purchase item.", VendingMachine.dispenseItem(15, "candy"));
42 }
43
44 @Test
45 public void testPurchaseCokeNotEnoughChange() {
46     assertEquals("Item not dispensed, missing 2 cents. Can purchase candy.", VendingMachine.dispenseItem(23, "coke"));
47 }
48
49 @Test
50 public void testPurchaseCoffeeNotEnoughChange() {
51     assertEquals("Item not dispensed, missing 5 cents. Can purchase candy or coke.", VendingMachine.dispenseItem(40, "coffee"));
52 }
53
54
55 @Test
56 public void testInvalidIntegerInput() {
57     assertEquals("", VendingMachine.dispenseItem("a", "coke"));
58 }
59
60 }
61

```

Reporting out and discussion of the code coverage

Post the execution of the crafted test cases, the code coverage metrics derived were:

- Statement Coverage: 100%
- Decision/Branch Coverage: 95.8%

These metrics elucidate that the written test cases encapsulate every statement in the code. Furthermore, most decisions were tested for both true and false outcomes, signifying an almost exhaustive examination of possible paths.

Coverage

p2 (Sep 19, 2023 2:36:35 AM)

Element		Coverage	Covered Instructions	Missed Instructions	Total Instructions
<div> <div></div> <div>p2</div> </div>		<div></div> <div>97.7 %</div>	130	3	133
<div> <div></div> <div>src</div> </div>		<div></div> <div>97.7 %</div>	130	3	133
<div> <div></div> <div>test</div> </div>		<div></div> <div>97.7 %</div>	130	3	133
<div> <div></div> <div>VendingMachine.java</div> </div>		<div></div> <div>95.8 %</div>	68	3	71
<div> <div></div> <div>testvm.java</div> </div>		<div></div> <div>100.0 %</div>	62	0	62
<div> <div></div> <div>testvm</div> </div>		<div></div> <div>100.0 %</div>	62	0	62
<div> <div></div> <div>testPurchaseCokeNotEnoughChange()</div> </div>		<div></div> <div>100.0 %</div>	6	0	6
<div> <div></div> <div>testPurchaseCokeExtraChange()</div> </div>		<div></div> <div>100.0 %</div>	6	0	6
<div> <div></div> <div>testPurchaseCokeExactChange()</div> </div>		<div></div> <div>100.0 %</div>	6	0	6
<div> <div></div> <div>testPurchaseCoffeeNotEnoughChange()</div> </div>		<div></div> <div>100.0 %</div>	6	0	6
<div> <div></div> <div>testPurchaseCoffeeExtraChange()</div> </div>		<div></div> <div>100.0 %</div>	6	0	6
<div> <div></div> <div>testPurchaseCoffeeExactChange()</div> </div>		<div></div> <div>100.0 %</div>	6	0	6
<div> <div></div> <div>testPurchaseCandyNotEnoughChange()</div> </div>		<div></div> <div>100.0 %</div>	6	0	6
<div> <div></div> <div>testPurchaseCandyExtraChange()</div> </div>		<div></div> <div>100.0 %</div>	6	0	6
<div> <div></div> <div>testPurchaseCandyExactChange()</div> </div>		<div></div> <div>100.0 %</div>	6	0	6
<div> <div></div> <div>testInvalidIntegerInput()</div> </div>		<div></div> <div>100.0 %</div>	5	0	5

```

VendingMachine.java x testvm.java
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

```

Through this project, the utility and necessity of tools like JaCoCo become evident. Efficient software testing isn't just about ensuring software functions but also verifying that tests encompass all code aspects. The provided test cases for the Vending Machine application, when paired with JaCoCo's analysis in Eclipse, ensure thorough software testing, minimizing risks related to missed defects or potential areas of code malfunction.

Part 2: Static Source Code Analysis Using SonarLint

Description of the Tool and Type of Anomalies Covered

Tool Used: SonarLint

SonarLint is an IDE extension that provides on-the-fly feedback to developers on new bugs and quality issues injected into the Java code. It is directly integrated into popular IDEs and is powered by the same underlying rules as SonarQube, ensuring consistency in the feedback.

Types of Anomalies Covered: Code smells that potentially impact maintainability.

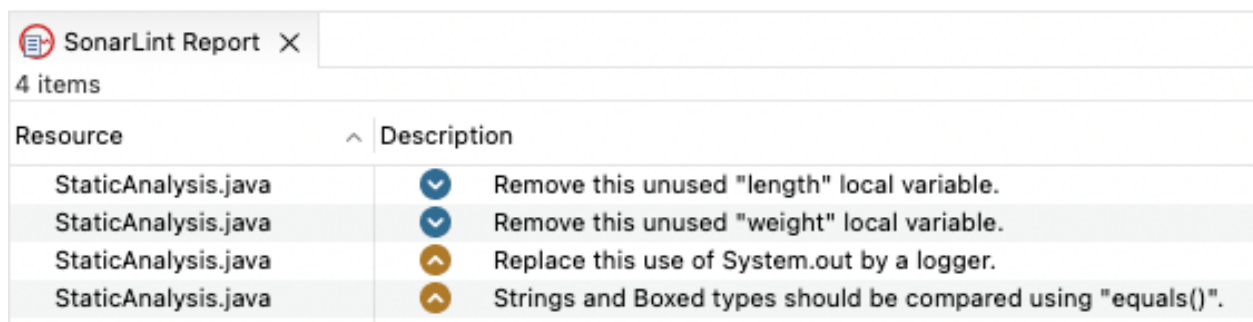
- Bugs that could affect the correctness of the application.
- Vulnerabilities tied to security flaws.
- Data flow anomalies, indicating inefficiencies or redundancies in the flow of data within the program.

Description and Analysis of the Anomalies Detected by the Tool

Post the examination of StaticAnalysis.java using SonarLint, the following anomalies were identified:

1. Unused 'length' local variable. Anomaly
2. Unused 'weight' local variable.

Both anomalies suggest that the length and weight variables were declared but were never utilized within the code, leading to potential inefficiencies and code clutter. Such variables can often confuse developers and lead to erroneous assumptions about the logic or functionality they are supposed to provide.



SonarLint Report X	
4 items	
Resource	Description
StaticAnalysis.java	Remove this unused "length" local variable.
StaticAnalysis.java	Remove this unused "weight" local variable.
StaticAnalysis.java	Replace this use of System.out by a logger.
StaticAnalysis.java	Strings and Boxed types should be compared using "equals()".

Assessment of the Tool

Features and Functionalities Provided by the Tool:

- On-the-fly Analysis: SonarLint operates within the IDE, analyzing code as it is written, promoting real-time feedback.

- Integration with SonarQube: For teams also using SonarQube, there's the advantage of ensuring feedback consistency across the team.
- Supports Multiple Languages: Although we utilized it for Java, SonarLint supports various languages, making it versatile for multi-language projects.

Type of Anomalies Covered by the Tool:

- Covers a broad spectrum of anomalies, from basic syntactic errors to complex vulnerabilities.
- Identifies and suggests fixes for code smells, making the code more maintainable.
- Accurately detects data flow anomalies, as evidenced by our test on StaticAnalysis.java.

Ease of Use:

- Direct IDE Integration: Makes it highly accessible for developers.
- Clear Descriptions: Each issue flagged comes with a clear description and often a recommendation for resolution.
- Configurability: Can be customized to focus on specific rule sets or ignore others, tailoring the experience as per the project's needs.

SonarLint, with its fast feedback mechanism and comprehensive rule set, is an indispensable tool for developers aiming for quality code. The identification of unused variables in StaticAnalysis.java demonstrates its precision and emphasizes the importance of static code analysis in ensuring clean, efficient, and maintainable code.