

CSE 565 Project 2

September 15, 2020

Brando Flores

Table of Contents

[1 Part 1 1](#_Toc82960438)

[1.1 Purpose, Goal, and Tools Used 1](#_Toc82960439)

[1.2 Set of Test Cases 1](#_Toc82960440)

[1.3 Coverage Achieved 1](#_Toc82960441)

[1.4 Tool Usefulness 2](#_Toc82960442)

[2 Part 2 3](#_Toc82960443)

[2.1 Tool Used 3](#_Toc82960444)

[2.2 Data Flow Anomalies 3](#_Toc82960445)

[2.3 Analysis Performed 3](#_Toc82960446)

[2.4 Tool Usefulness 4](#_Toc82960447)

# Part 1

## Purpose, Goal, and Tools Used

The purpose of this project was to investigate tools that aided in the determining statement (line) and decision (branch) coverage of code. The code was a simple Java class representing a vending machine that dispensed candy, coke or coffee and returned change to the user. The goal of this project was to develop test cases that resulted in 100% statement coverage and at least 90% decision coverage. The IntelliJ IDE was used to run the Java file and used for developing Junit test cases. These test cases were evaluated with the JaCoCo plugin to determine statement and branch coverage.

## Set of Test Cases

The following is the Junit test cases used to track the statement and branch coverage. Text

Description automatically generated

## Coverage Achieved

The following shows the statement (line) and decision (branch) coverage depicted in the program.

Graphical user interface, application

Description automatically generated

## Tool Usefulness

I have determined that the IntelliJ tool combined with the JaCoCo plugin is an extremely useful tool. The following screenshot highlights where the code is being hit. This is particularly useful in helping to debug code that is not being hit, reducing redundant code, and reducing the amount of tests needed to guarantee accuracy of code.

Text

Description automatically generated

# Part 2

## Tool Used

The tools used for this portion of the project are largely the same as in part 1. The IntelliJ IDE is still used for the static analysis of the provided class. The static analysis was performed by right clicking on the StaticAnalysis class, selecting Analyze from the dropdown, then Inspect Code.

## Data Flow Anomalies

The tool found that that there were two different types of data flow anomalies. The two types were unused declaration and unused assignment. The unused declaration occurs when the code defines the StaticAnalysis constructor. In the constructor there are two variables being defined: weight and length. Although these variables are defined and values assigned, they are never used in the class and cannot be used because they remain in the scope of the constructor. This constructor should be removed so that the class' default constructor may be used and the unused variables removed. The second anomaly, the unused assignment, occurs in the calculateCost function. The cost and output variables are initialized with values, but those values are never used before they are reassigned again. This makes the initial assignment redundant, and it should be removed.

## Analysis Performed

The following shows the static analysis perform

Text

Description automatically generated

## Tool Usefulness

I believe that this tool is incredibly useful because it catches syntax problems that may be looked over by the programmer. Routinely performing this analysis can help to minimize bugs and clean up code. Although, one point of feedback I would provide is that many IDEs provide this functionality build in, and as the programmer is writing code. IDEs such as Android Studio and Visual Studio for Mac routinely check for the redundant assignment of variables and unused variables.