

CSE 565 - Software Verification Validation and Testing

Assignment 4

Decision Code Coverage

Kedar Sai Nadh Reddy Kanchi

School of Computing and Augmented Intelligence
Arizona State University, Tempe
kkanchi@asu.edu

IntelliJ IDEA Code Coverage Runner Tool

IntelliJ IDEA is an Integrated Development Environment (IDE) for Java, Kotlin, and other programming languages. One of its features is the IntelliJ Code Coverage Runner, an inbuilt code coverage tool that allows developers to measure the extent to which their tests exercise their code[3 & 7].

Scope of the tool

The scope of the Code Coverage Runner tool in IntelliJ IDEA includes the ability to measure code coverage for a wide variety of programming languages and frameworks, including Java, Kotlin, Groovy, Scala, JavaScript, TypeScript, and many others. Code coverage for unit tests, integration tests, and functional tests can be analyzed using the tool[3 & 7].

Purpose of the tool

The purpose of the Code Coverage Runner tool is to analyze the code and determine which statements or lines have been executed during the tests, providing developers with insights into how thoroughly their code has been tested. Code coverage assists developers in identifying untested or under-tested areas of their code and creating more effective tests to improve code quality[3 & 7].

Example Usage

One example of a use case for this tool is when a software development team wants to ensure that their code has been thoroughly tested before deployment. The team can identify areas that require additional testing and improve the overall quality of the software by running the code coverage tool.

For instance, suppose a team is working on a banking application, and they want to make sure that their code is adequately tested. Using the IntelliJ IDEA code coverage runner tool, they can produce a report on the level of code coverage attained by their unit tests. If the coverage is inadequate, the team can write additional tests to cover the gaps. By doing this, they can make sure their application is solid and dependable, which will make production bugs and errors less likely.

Types of code coverage the tool provides

The code coverage runner tool in IntelliJ IDEA offers various types of code coverage to assist developers in determin-

ing how thoroughly a given test suite executes a program's source code.

The following are the types of code coverage analysis provided by the IntelliJ IDEA code coverage runner tool:

- **Statement/Line Coverage:** This measures the percentage of statements/lines that are executed during testing. A statement is a single line of code that performs an action, such as an assignment or a function call. A statement is considered covered if it is executed at least once during testing[1 & 2].
- **Branch/Decision Coverage:** This measures the percentage of branches in the code that are executed during testing. A branch is a decision point in the code where the program can take one of two or more paths. A branch is considered covered if both the true and false branches are executed at least once during testing[1 & 2].
- **Method Coverage :** This measures the percentage of methods that are executed during testing. A method is a collection of statements that performs a specific task. A method is considered covered if it is executed at least once during testing[1 & 2].
- **Class Coverage :** This measures the percentage of classes that are executed during testing. A class is a blueprint for creating objects that have a specific set of properties and behaviors. A class is considered covered if all of its methods are executed at least once during testing[1 & 2].

Test Case Description, Scope and Coverage

Before delving into each test case in depth, we can see that the lines numbered 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 19, 20, 23, 24, 25, 28, 29, 30, 38, 39, 40, 41, 42, and 43 are always executed regardless of which test case is executed in the given java code file. This is due to the fact that some of the lines that are not part of the decision or condition blocks are categorized as statements that define classes or methods, use syntax, or are part of common code.

To summarize, there are 24 lines/statements and 8 decisions/conditions in the dispenseItem function. The 8 decisions/conditions result in a total of 16 branches because each decision has two possible outcomes (true or false). The 8 decisions/conditions in the code are:

- **Decision 1:** item == "candy"
- **Decision 2:** item == "coke"

- **Decision 3:** item == "coffee"
- **Decision 4:** input > cost
- **Decision 5:** input == cost
- **Decision 6:** input < 45
- **Decision 7:** input < 25
- **Decision 8:** input < 20

Test Case 1

Description

In this test case, the parameters passed into the dispenseItem function are, **input** = 40 and **item** = "candy". The **expected result** is "Item dispensed and change of 20 returned".

Scope

The scope of this test case is to determine whether the code is capable of returning the selected product and any remaining change.

Coverage

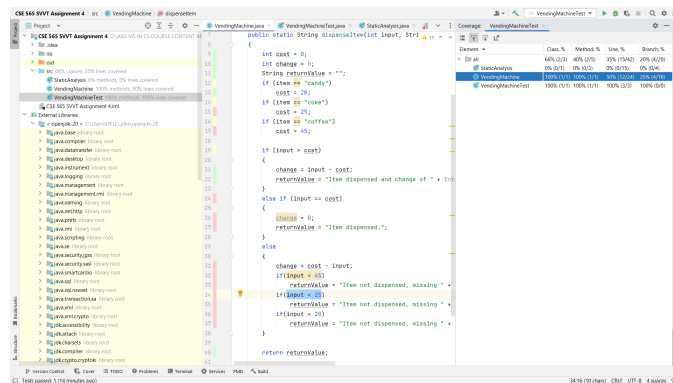


Figure 1: Code Coverage by Test Case 1

Statement/Line Coverage: As shown in Figure 1, Test Case 1 covered the following 12 lines: 9, 10, 11, 12, 13, 14, 16, 19, 21, 22, and 40, accounting for half(50%) of the total number of lines.

Decision/Branch Coverage: As shown in Figure 1, Test Case 1 has covered the four conditions and four branches:

- **Decision 1:** item == "candy" - The branch covered for this decision is **True** - because the value for the parameter "item" passed into this function definition from the test case was "candy".
- **Decision 2:** item == "coke" - The branch covered for this decision is **False** - because the value for the parameter "item" passed into this function definition from the test case was "candy".
- **Decision 3:** item == "coffee" - The branch covered for this decision is **False** - because the value for the parameter "item" passed into this function definition from the test case was "candy".
- **Decision 4:** input > cost - The branch covered for this decision is **True** - because the value for the variable "cost"

is set to 20 when the line 13 was executed and since the the value for the parameter "input" passed into this function definition by from the test case was 40 - which makes condition defined in line 19 evaluate to True.

Test Case 2

Description

In this test case, the parameters passed into the dispenseItem function are, **input** = 25 and **item** = "coke". The **expected result** is "Item dispensed".

Scope

The scope of this test case to check one of the main functionalities that is whether the code is able to return the selected product when there is no remaining change.

Coverage

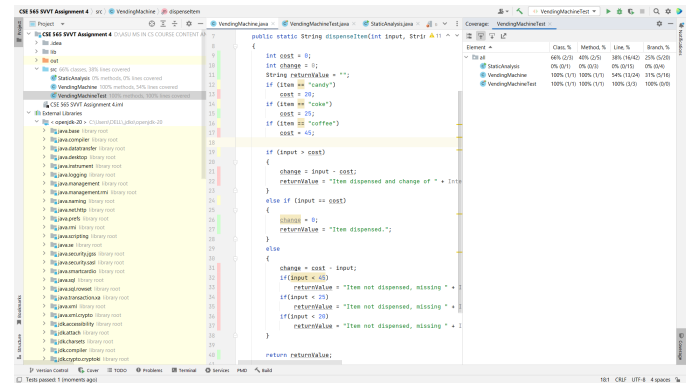


Figure 2: Code Coverage by Test Case 2

Statement/Line Coverage: So, from the above image, Figure 2, you can see that the Test Case 2 has covered the following 13 lines - 9, 10, 11, 12, 14, 15, 16, 19, 24, 26, 27, and 40 - which is (54%) of the total number of lines.

Decision/Branch Coverage: So, again from the above image, Figure 2, you can see that the Test Case 2 has covered the following 5 conditions and 5 branches:

- **Decision 1:** item == "candy" - The branch covered for this decision is **False** - because the value for the parameter "item" passed into this function definition by from the test case was "coke".
- **Decision 2:** item == "coke" - The branch covered for this decision is **True** - because the value for the parameter "item" passed into this function definition by from the test case was "coke".
- **Decision 3:** item == "coffee" - The branch covered for this decision is **False** - because the value for the parameter "item" passed into this function definition by from the test case was "coke".
- **Decision 4:** input > cost - The branch covered for this decision is **False** - because the value for the variable "cost" is set to 25 when the line 15 [cost = 25;] was executed and since the the value for the parameter "input" passed into this function definition by from the test case was 25 - which makes condition defined in line 19 evaluate to False.

- **Decision 5:** `input == cost` - The branch covered for this decision is **True** - because the value for the variable "cost" is set to 25 when the line 15 [`cost = 25;`] was executed and since the value for the parameter "input" passed into this function definition by from the test case was 25 - which makes condition defined in line 24 evaluate to True.

Test Case 3

Description

In this test case, the parameters passed into the `dispenseItem` function are, **input** = 40 and **item** = "coffee". The **expected result** is "Item not dispensed, missing 5 cents. Can purchase candy or coke".

Scope

The scope of this test case to check one of the main functionalities that is whether the code is able to displays the amount necessary to buy the product and other products to purchase when there is not enough money to buy the product that the user has selected.

Coverage

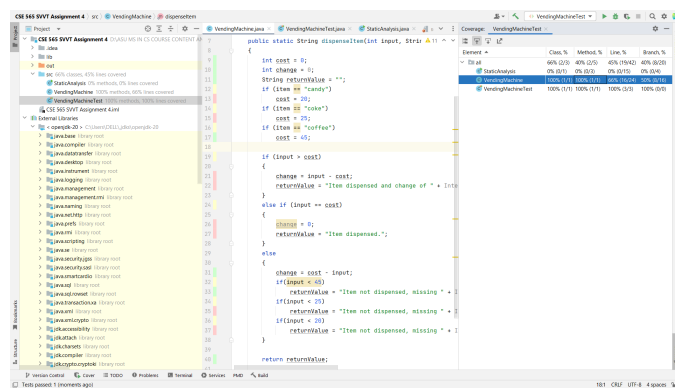


Figure 3: Code Coverage by Test Case 3

Statement/Line Coverage: So, from the above image, Figure 3, you can see that the Test Case 2 has covered the following 16 lines - 9, 10, 11, 12, 14, 16, 17, 19, 24, 31, 32, 33, 34, 36, and 40 - which is (66%) of the total number of lines.

Decision/Branch Coverage: So, again from the above image, Figure 3, you can see that the Test Case 3 has covered the following 8 conditions and 8 branches:

- **Decision 1:** `item == "candy"` - The branch covered for this decision is **False** - because the value for the parameter "item" passed into this function definition by from the test case was "coffee".
- **Decision 2:** `item == "coke"` - The branch covered for this decision is **False** - because the value for the parameter "item" passed into this function definition by from the test case was "coffee".
- **Decision 3:** `item == "coffee"` - The branch covered for this decision is **True** - because the value for the parameter "item" passed into this function definition by from the test case was "coffee".

- **Decision 4:** `input > cost` - The branch covered for this decision is **False** - because the value for the variable "cost" is set to 45 when the line 17 [`cost = 45;`] was executed and since the value for the parameter "input" passed into this function definition by from the test case was 40 - which makes condition defined in line 19 evaluate to False.

- **Decision 5:** `input == cost` - The branch covered for this decision is **False** - because the value for the variable "cost" is set to 45 when the line 17 [`cost = 45;`] was executed and since the value for the parameter "input" passed into this function definition by from the test case was 40 - which makes condition defined in line 24 evaluate to False.

- **Decision 6:** `input < 45` - The branch covered for this decision is **True** - because the value for the parameter "input" passed into this function definition by from the test case was 40 - which makes condition defined in line 32 evaluate to True.

- **Decision 7:** `input < 25` - The branch covered for this decision is **False** - because the value for the parameter "input" passed into this function definition by from the test case was 40 - which makes condition defined in line 34 evaluate to False.

- **Decision 8:** `input < 20` - The branch covered for this decision is **False** - because the value for the parameter "input" passed into this function definition by from the test case was 40 - which makes condition defined in line 36 evaluate to False.

Test Case 4

Description

In this test case, the parameters passed into the `dispenseItem` function are, **input** = 22 and **item** = "coke". The **expected result** is "Item not dispensed, missing 3 cents. Can purchase candy".

Scope

The scope of this test case to check one of the main functionalities that is whether the code is able to displays the amount necessary to buy the product and other products to purchase when there is not enough money to buy the product that the user has selected.

Coverage

Statement/Line Coverage: So, from the image, Figure 4, you can see that the Test Case 2 has covered the following 17 lines - 9, 10, 11, 12, 14, 15, 16, 19, 24, 31, 32, 33, 34, 35, 36, and 40 - which is (70%) of the total number of lines.

Decision/Branch Coverage: So, again from the image, Figure 4, you can see that the Test Case 4 has covered the following 8 conditions and 8 branches out of a total of 16 branches. The decisions and branches covered with this test case are:

- **Decision 1:** `item == "candy"` - The branch covered for this decision is **False** - because the value for the parameter "item" passed into this function definition by from the test case was "coke".

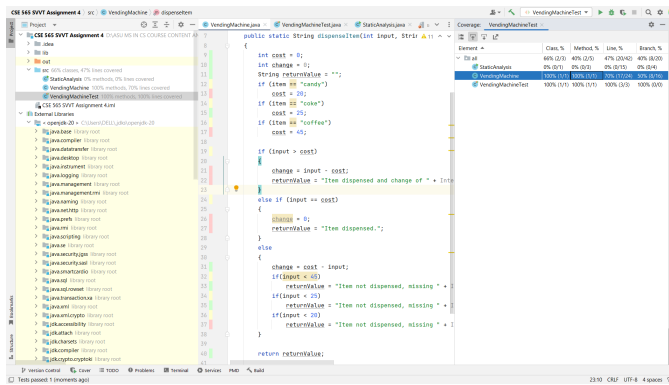


Figure 4: Code Coverage by Test Case 4

- **Decision 2:** item == "coke" - The branch covered for this decision is **True** - because the value for the parameter "item" passed into this function definition by from the test case was "coke".
- **Decision 3:** item == "coffee" - The branch covered for this decision is **False** - because the value for the parameter "item" passed into this function definition by from the test case was "coke".
- **Decision 4:** input > cost - The branch covered for this decision is **False** - because the value for the variable "cost" is set to 25 when the line 15 [cost = 25;] was executed and since the the value for the parameter "input" passed into this function definition by from the test case was 22 - which makes condition defined in line 19 evaluate to False.
- **Decision 5:** input == cost - The branch covered for this decision is **False** - because the value for the variable "cost" is set to 25 when the line 15 [cost = 25;] was executed and since the the value for the parameter "input" passed into this function definition by from the test case was 22 - which makes condition defined in line 24 evaluate to False.
- **Decision 6:** input < 45 - The branch covered for this decision is **True** - because the value for the parameter "input" passed into this function definition by from the test case was 22 - which makes condition defined in line 32 evaluate to True.
- **Decision 7:** input < 25 - The branch covered for this decision is **True** - because the value for the parameter "input" passed into this function definition by from the test case was 22 - which makes condition defined in line 34 evaluate to True.
- **Decision 8:** input < 20 - The branch covered for this decision is **False** - because the value for the parameter "input" passed into this function definition by from the test case was 22 - which makes condition defined in line 36 evaluate to False.

Test Case 5

Description

In this test case, the parameters passed into the dispense function are, **input** = 18 and **item** = "candy". The **expected result** is "Item not dispensed, missing 2 cents. Cannot purchase item".

Scope

The scope of this test case to check one of the main functionalities that is whether the code is able to only display the amount necessary to buy the product and no other suggestions when there is not enough money to buy the product that the user has selected or any other product that is available within the amount the use ha entered.

Coverage

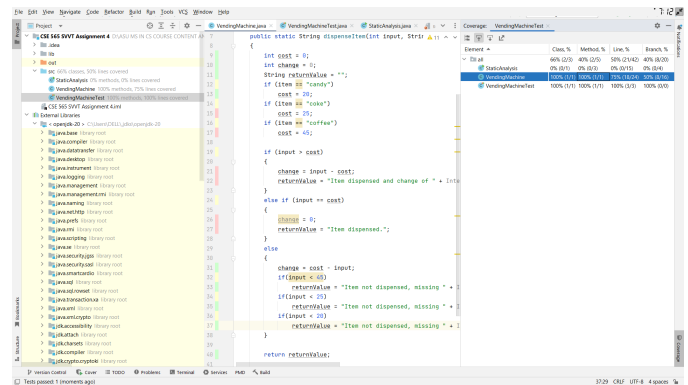


Figure 5: Code Coverage by Test Case 5

Statement/Line Coverage: So, from the image, Figure 5, you can see that the Test Case 2 has covered the following 18 lines - 9, 10, 11, 12, 13, 14, 16, 19, 24, 31, 32, 33, 34, 35, 36, 37, and 40 - which is (75%) of the total number of lines.

Decision/Branch Coverage: So, again from the image, Figure 5, you can see that the Test Case 5 has covered the following 8 conditions and 8 branches out of a total of 16 branches. The decisions and branches covered with this test case are:

- **Decision 1:** item == "candy" - The branch covered for this decision is **True** - because the value for the parameter "item" passed into this function definition by from the test case was "candy".
- **Decision 2:** item == "coke" - The branch covered for this decision is **False** - because the value for the parameter "item" passed into this function definition by from the test case was "candy".
- **Decision 3:** item == "coffee" - The branch covered for this decision is **False** - because the value for the parameter "item" passed into this function definition by from the test case was "candy".
- **Decision 4:** input > cost - The branch covered for this decision is **False** - because the value for the variable "cost" is set to 20 when the line 13 [cost = 20;] was executed and since the the value for the parameter "input" passed into this function definition by from the test case was 18

- which makes condition defined in line 19 evaluate to False.

- **Decision 5:** $\text{input} == \text{cost}$ - The branch covered for this decision is **False** - because the value for the variable "cost" is set to 20 when the line 13 [$\text{cost} = 20;$] was executed and since the the value for the parameter "input" passed into this function definition by from the test case was 18 - which makes condition defined in line 24 evaluate to False.
- **Decision 6:** $\text{input} < 45$ - The branch covered for this decision is **True** - because the value for the parameter "input" passed into this function definition by from the test case was 18 - which makes condition defined in line 32 evaluate to True.
- **Decision 7:** $\text{input} < 25$ - The branch covered for this decision is **True** - because the value for the parameter "input" passed into this function definition by from the test case was 18 - which makes condition defined in line 34 evaluate to True.
- **Decision 8:** $\text{input} < 20$ - The branch covered for this decision is **True** - because the value for the parameter "input" passed into this function definition by from the test case was 18 - which makes condition defined in line 36 evaluate to True.

Summary of the Tool's Coverage

Please see the attached zip file for the VendingMachineTest.java file containing the developed unit test cases. I created the minimum number of test cases which is 5, to complete the code coverage while satisfying the given condition mentioned in the project description - 100% statement coverage and at least 90% decision coverage. And, as shown in the attached images, Figures 8 and 9, the developed unit test cases provide the required coverage.

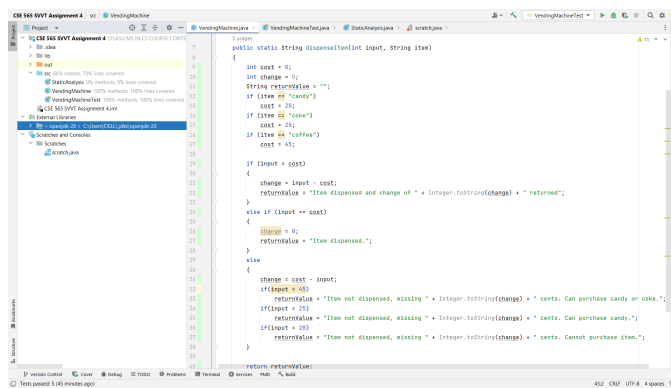


Figure 6: The visual representation of the code coverage

Total Statement Coverage

From Figure 6, you can see that we have a 100% statement coverage.

- **Line 9:** Covered by the Test Cases - 1, 2, 3, 4, and 5.

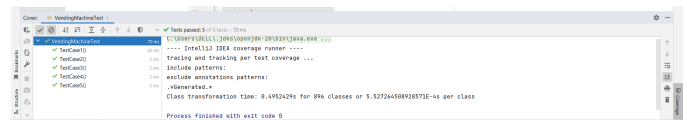


Figure 7: Terminal Output of the Total Code Coverage

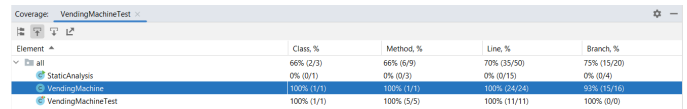


Figure 8: Total Code Coverage Statistics

- **Line 10:** Covered by the Test Cases - 1, 2, 3, 4, and 5.
- **Line 11:** Covered by the Test Cases - 1, 2, 3, 4, and 5.
- **Line 12:** Covered by the Test Cases - 1, 2, 3, 4, and 5.
- **Line 13:** Covered by the Test Cases - 1, and 5.
- **Line 14:** Covered by the Test Cases - 1, 2, 3, 4, and 5.
- **Line 15:** Covered by the Test Cases - 2, and 4.
- **Line 16:** Covered by the Test Cases - 1, 2, 3, 4, and 5.
- **Line 17:** Covered by the Test Cases - 3.
- **Line 19:** Covered by the Test Cases - 1, 2, 3, 4, and 5.
- **Line 21:** Covered by the Test Cases - 1.
- **Line 22:** Covered by the Test Cases - 1.
- **Line 24:** Covered by the Test Cases - 2, 3, 4, and 5.
- **Line 26:** Covered by the Test Cases - 2.
- **Line 27:** Covered by the Test Cases - 2.
- **Line 31:** Covered by the Test Cases - 3, 4, and 5.
- **Line 32:** Covered by the Test Cases - 3, 4, and 5.
- **Line 33:** Covered by the Test Cases - 3, 4, and 5.
- **Line 34:** Covered by the Test Cases - 3, 4, and 5.
- **Line 35:** Covered by the Test Cases - 4, and 5.
- **Line 36:** Covered by the Test Cases - 3, 4, and 5.
- **Line 37:** Covered by the Test Cases - 5.
- **Line 40:** Covered by the Test Cases - 1, 2, 3, 4, and 5.

Total Decision Coverage

Figure 6 shows that we have a decision coverage of 93%. This is due to the fact that, as stated in the project description, all decisions have been covered with the exception of the False branch for the decision/condition on line 32 ($\text{input} < 45$). So, the code covers 15 of the 16 branches (8 conditions), for a total of 93.75% coverage.

- **Decision 1:** $\text{item} == \text{"candy"}$
 - **True Branch:** Covered by the Test Cases - 1 and 5.
 - **False Branch:** Covered by the Test Cases - 2, 3, and 4.
- **Decision 2:** $\text{item} == \text{"coke"}$
 - **True Branch:** Covered by the Test Cases - 2 and 4.

Current scope: all classes - <empty package name>
Coverage Summary for Package: <empty package>

| Package | Class % | Method % | Branch % | Line % |
|-----------------|-------------|--------------|-------------|-------------|
| <empty package> | 66.7% (2/3) | 72.7% (8/11) | 75% (15/20) | 70% (35/50) |

| Class | Class % | Method % | Branch % | Line % |
|----------------------|------------|------------|--------------|--------------|
| StaticAnalysis | 0% (0/1) | 0% (0/3) | 0% (0/5) | 0% (0/3) |
| TestingInterface | 100% (1/1) | 100% (2/2) | 10.0% (1/10) | 100% (24/24) |
| TestingInterfaceImpl | 100% (1/1) | 100% (3/3) | 100% (15/15) | 100% (21/21) |

generated on 2023-03-22 20:30

Figure 9: Exported Coverage Report

- **False Branch:** Covered by the Test Cases - 1, 3, and 5.
- **Decision 3:** item == "coffee"
 - **True Branch:** Covered by the Test Cases - 3.
 - **False Branch:** Covered by the Test Cases - 1, 2, 4, and 5.
- **Decision 4:** input > cost
 - **True Branch:** Covered by the Test Cases - 1.
 - **False Branch:** Covered by the Test Cases - 2, 3, 4, and 5.
- **Decision 5:** input == cost
 - **True Branch:** Covered by the Test Cases - 2.
 - **False Branch:** Covered by the Test Cases - 3, 4, and 5.
- **Decision 6:** input < 45
 - **True Branch:** Covered by the Test Cases - 3, 4, and 5.
 - **False Branch:** NOT COVERED.
- **Decision 7:** input < 25
 - **True Branch:** Covered by the Test Cases - 4.
 - **False Branch:** Covered by the Test Cases - 3 and 5.
- **Decision 8:** input < 20
 - **True Branch:** Covered by the Test Cases - 5.
 - **False Branch:** Covered by the Test Cases - 3 and 4.

Evaluation of the tool’s usefulness

The IntelliJ IDEA code coverage runner is a useful tool for developers who want to improve the quality of their code by ensuring adequate test coverage. Its user-friendly interface, robust coverage analysis capabilities, and useful features make it a valuable addition to any development workflow.

Usability

In terms of usability, the IntelliJ IDEA code coverage runner is well-known for being user-friendly and easy to navigate. It provides a simple and intuitive interface for configuring code coverage settings and generating coverage reports and identifying areas that need improvement. The tool also integrates seamlessly with IntelliJ IDEA, making it an obvious choice for developers who use the IDE.

Coverage

In terms of coverage, the IntelliJ IDEA code coverage runner provides comprehensive coverage analysis for Java code, allowing developers to determine the proportion of code that was executed during testing. It can track code coverage at the

class, method, and line levels and generate reports indicating which sections of the code were executed during testing. This data can be used to identify areas of code that are not being adequately tested and may contain bugs.

Features

In terms of features, the IntelliJ IDEA code coverage runner provides several useful capabilities that can aid developers in improving their code coverage. For example:

- It supports a variety of testing frameworks, including JUnit, TestNG, and Cucumber, among others.
- It allows developers to perform coverage analysis on specific classes or packages, allowing them to concentrate their efforts on specific areas of the code-base.
- It can also indicate which parts of the code were not tested by highlighting code lines that were not executed during testing.
- Additionally, it has a feature that makes it simpler to make sure that all pertinent tests are run throughout the development process. This feature enables you to run tests that are impacted by code changes.
- It integrates with build tools such as Gradle and Maven, allowing developers to integrate coverage analysis into their development process.
- To reduce noise in the coverage report, it can be configured to exclude a set of classes or methods from analysis.
- The code coverage runner can generate coverage reports in different formats, including HTML, XML, or CSV, and can be integrated with continuous integration (CI) tools like Jenkins or TeamCity.

Part 2 - Static Source Code Analysis Tool

PMD(Programming Mistake Detector) is the tool used for this assignment. PMD is well-known in the Java community and is available as a plugin for several popular Java development environments, such as Eclipse and IntelliJ IDEA[5].

PMD is a well-known open-source static source code analysis tool for detecting potential bugs or issues in programming languages like Java, Kotlin, and others. Without running the program, PMD examines its source code for common programming errors, identifying coding issues such as unused variables, empty catch blocks, and inefficient code constructs and recommending improvements.

To perform its analysis, PMD employs a set of predefined rules or code guidelines that cover a wide range of coding issues, such as inefficient code, unused variables, and potential security vulnerabilities. It can also be customized with additional rules to meet the requirements of a specific project. These rules are based on best practices and common coding errors, and they can be tailored to the needs of a specific project. When PMD is run, it reports any violations of these rules, as well as where they occur in the source code[6].

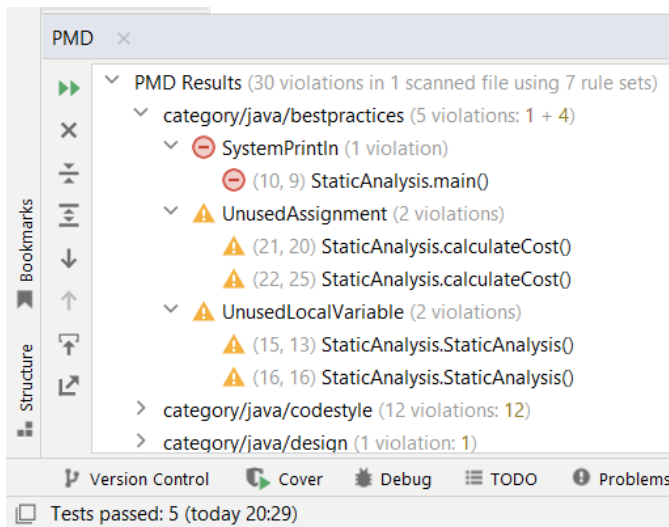


Figure 10: PMD RESULTS 1

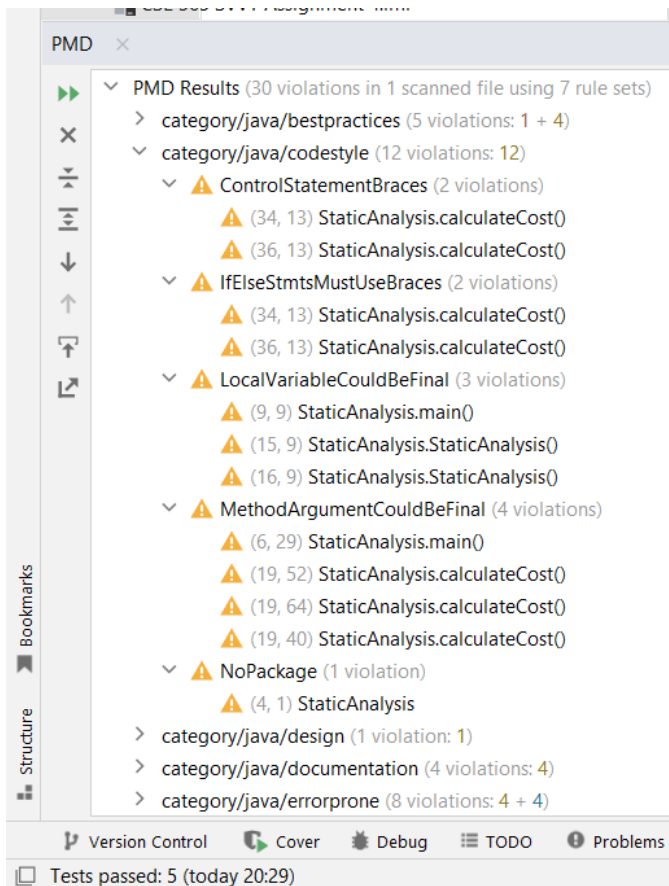


Figure 11: PMD RESULTS 2

Output from the PMD tool

The dataflow analysis tracks local definitions, undefinitions, and references to variables along the data flow's various

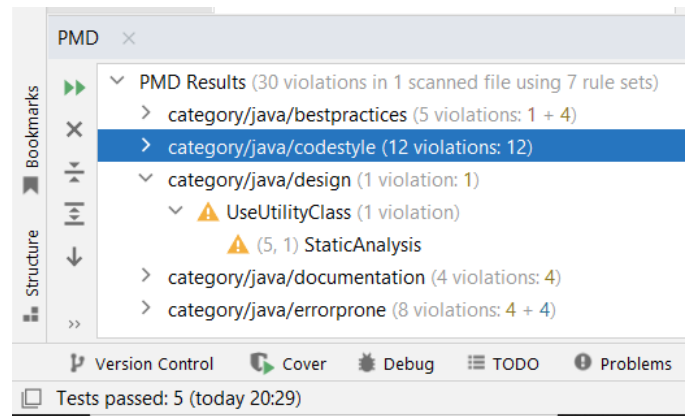


Figure 12: PMD RESULTS 3

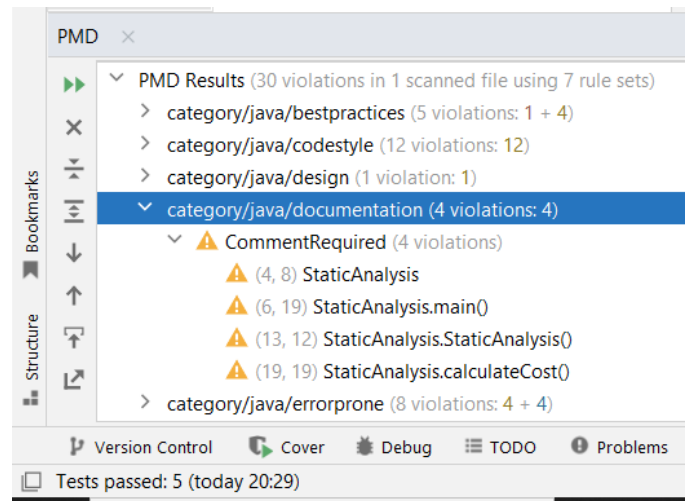


Figure 13: PMD RESULTS 4

paths. Various problems can be identified using these data [5]. There are two types of data anomalies:

- **DU - Anomaly:** A recently defined variable is undefined. These anomalies may appear in normal source text.
- **DD - Anomaly:** A recently defined variable is redefined. This is ominous but don't have to be a bug.

Figure 14 shows that there is a sub section called DataFlowAnomalyAnalysis under the category/java/errorprone section. However, according to the official documentation, this rule is now deprecated, and we should instead use the results from the UnusedAssignment part in the category/java/bestpractices section, as shown in Figure 10.

Description of the two data flow anomalies

Therefore, the StaticAnalysis.java file contains the following two different data flow anomalies:

- **UnusedAssignment** – `int cost = 0;` – Found 'DD'-anomaly for variable 'cost' (lines '21'-'26') in public static String calculateCost(). As the value of cost = 0 is

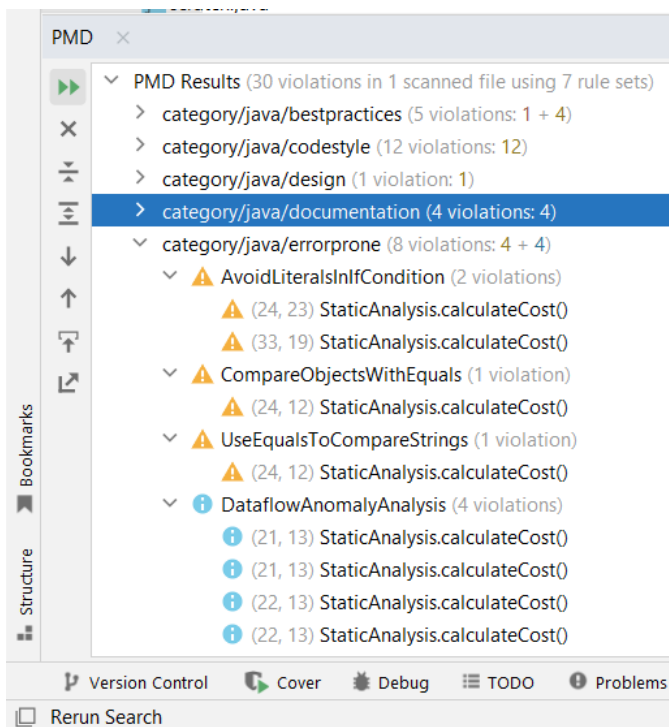


Figure 14: PMD RESULTS 5

never used and also it is later redefined on lines 26 and 30 depending on how the decision on line 24 evaluates, this error is thrown as Unused Assignment.

- **UnusedAssignment – String output = "";** – Found 'DD'-anomaly for variable 'output' (lines '22'-'34') in public static String calculateCost(). As the value of output = "" is never used and also it is later redefined to a different string value on lines 34 and 36 depending on how the decision on line 33 evaluates, this error is thrown as Unused Assignment.

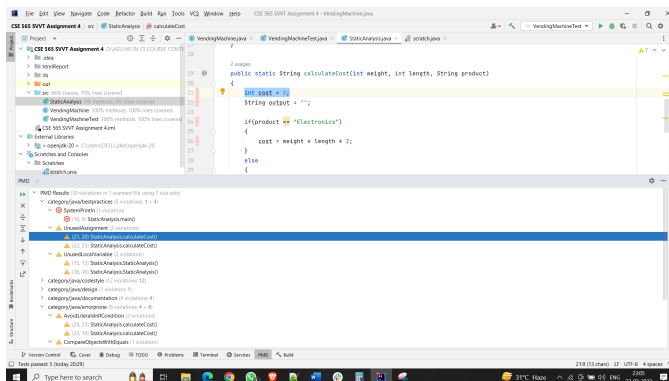


Figure 15: Data Anomaly 1 - UnusedAssignment - int cost = 0;

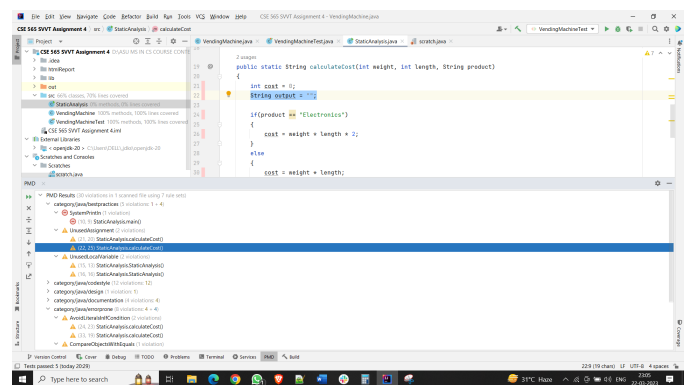


Figure 16: Data Anomaly 2 - UnusedAssignment - String output = "";

Evaluation of the PMD tool's usefulness

PMD (Programming Mistake Detector) is a free and open-source static code analysis tool for detecting common programming errors in Java, JavaScript, and other programming languages. PMD can detect problems like unused variables, unused parameters, and inefficient code[5].

Usability

PMD is relatively easy to use, with a simple command-line interface that can be easily integrated with various popular Integrated Development Environments (IDEs) such as Eclipse, IntelliJ IDEA, and NetBeans. It is also available as a plugin for Maven, Gradle, and Ant. PMD has an intuitive user interface and is appropriate for both novice and seasoned developers. PMD comes with comprehensive instructions on how to use the tool and interpret the results.

Coverage

PMD comes with a wide range of rules for detecting various types of programming errors. The software development tool addresses a wide range of issues, including code complexity, unused code, error-prone constructs, and inefficient code. It includes over 400 built-in rules for detecting common programming errors and can be supplemented with custom rules. Custom rules can also be added to PMD, allowing developers to tailor the tool to their specific needs. However, on the other hand, PMD's coverage is limited to Java and a few other programming languages.

Features

PMD provides various features that make it a useful tool for software development. Some of the notable features include:

- **Customizable rulesets:** PMD allows developers to create their own rulesets, enabling them to target specific areas of their codebase.
- **XML report format:** PMD generates an XML report that can be easily parsed by other tools or integrated into a continuous integration (CI) process.

- **Automated code analysis:** PMD can be run as part of a continuous integration (CI) process to automatically detect programming mistakes in code changes.
- **Automatic code fixing:** PMD can automatically resolve some detected issues, such as removing unused variables or importing missing classes.
- **Support for multiple languages:** PMD includes plugins for other programming languages such as PHP, Apex, and PL/SQL.


References

- [Prof. James Collofello, CSE 565 Lecture Videos, 2023](#)
- [Prof. Ayca Tuzmen, CSE 565 Lectures, 2023](#)
- [Code coverage](#)
- [Collecting Unit Test Coverage in IntelliJ IDEA](#)
- [PMD Source Code Analyzer Project](#)
- [How To Use PMD Code Analyzer With IntelliJ](#)
- [Code Coverage](#)

Current scope: [all classes](#) | <empty package name>

Coverage Summary for Package: <empty package>

| Package | Class, % | Method, % | Branch, % | Line, % |
|-----------------|-------------|--------------|-------------|-------------|
| <empty package> | 66.7% (2/3) | 72.7% (8/11) | 75% (15/20) | 70% (35/50) |

| Class  | Class, % | Method, % | Branch, % | Line, % |
|---|------------|------------|---------------|--------------|
| StaticAnalysis | 0% (0/1) | 0% (0/3) | 0% (0/4) | 0% (0/15) |
| VendingMachine | 100% (1/1) | 100% (2/2) | 93.8% (15/16) | 100% (24/24) |
| VendingMachineTest | 100% (1/1) | 100% (6/6) | | 100% (11/11) |

generated on 2023-03-22 20:30

PMD report

Problems found

| # | File | Line | Problem |
|----|---|------|--|
| 1 | D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 4 | All classes, interfaces, enums and annotations must belong to a named package |
| 2 | D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 4 | Class comments are required |
| 3 | D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 5 | All methods are static. Consider using a utility class instead. Alternatively, you could add a private constructor or make the class abstract to silence this warning. |
| 4 | D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 6 | Parameter 'args' is not assigned and could be declared final |
| 5 | D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 6 | Public method and constructor comments are required |
| 6 | D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 9 | Local variable 'value' could be declared final |
| 7 | D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 10 | System.out.println is used |
| 8 | D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 13 | Public method and constructor comments are required |
| 9 | D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 15 | Avoid unused local variables such as 'weight'. |
| 10 | D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING | 15 | Local variable 'weight' could be declared final |

2023\CSE 565 Software Verification Validation and
Testing\ASSIGNMENTS\Assignment 4 - Decision Code
Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java

D:\ASU MS IN CS COURSE CONTENT AND
ALL\SEMESTERS COURSES CONTENT\SPRING

11 2023\CSE 565 Software Verification Validation and
Testing\ASSIGNMENTS\Assignment 4 - Decision Code
Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java

16 [Avoid unused local variables
such as 'length'.](#)

D:\ASU MS IN CS COURSE CONTENT AND
ALL\SEMESTERS COURSES CONTENT\SPRING

12 2023\CSE 565 Software Verification Validation and
Testing\ASSIGNMENTS\Assignment 4 - Decision Code
Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java

16 [Local variable 'length' could be
declared final](#)

D:\ASU MS IN CS COURSE CONTENT AND
ALL\SEMESTERS COURSES CONTENT\SPRING

13 2023\CSE 565 Software Verification Validation and
Testing\ASSIGNMENTS\Assignment 4 - Decision Code
Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java

19 [Parameter 'length' is not assigned
and could be declared final](#)

D:\ASU MS IN CS COURSE CONTENT AND
ALL\SEMESTERS COURSES CONTENT\SPRING

14 2023\CSE 565 Software Verification Validation and
Testing\ASSIGNMENTS\Assignment 4 - Decision Code
Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java

19 [Parameter 'product' is not
assigned and could be declared
final](#)

D:\ASU MS IN CS COURSE CONTENT AND
ALL\SEMESTERS COURSES CONTENT\SPRING

15 2023\CSE 565 Software Verification Validation and
Testing\ASSIGNMENTS\Assignment 4 - Decision Code
Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java

19 [Parameter 'weight' is not assigned
and could be declared final](#)

D:\ASU MS IN CS COURSE CONTENT AND
ALL\SEMESTERS COURSES CONTENT\SPRING

16 2023\CSE 565 Software Verification Validation and
Testing\ASSIGNMENTS\Assignment 4 - Decision Code
Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java

19 [Public method and constructor
comments are required](#)

D:\ASU MS IN CS COURSE CONTENT AND
ALL\SEMESTERS COURSES CONTENT\SPRING

17 2023\CSE 565 Software Verification Validation and
Testing\ASSIGNMENTS\Assignment 4 - Decision Code
Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java

21 [Found 'DD'-anomaly for variable
'cost' \(lines '21'-'26'\).](#)

D:\ASU MS IN CS COURSE CONTENT AND
ALL\SEMESTERS COURSES CONTENT\SPRING

18 2023\CSE 565 Software Verification Validation and
Testing\ASSIGNMENTS\Assignment 4 - Decision Code
Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java

21 [Found 'DD'-anomaly for variable
'cost' \(lines '21'-'30'\).](#)

D:\ASU MS IN CS COURSE CONTENT AND
ALL\SEMESTERS COURSES CONTENT\SPRING

19 2023\CSE 565 Software Verification Validation and
Testing\ASSIGNMENTS\Assignment 4 - Decision Code
Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java

21 [The initializer for variable 'cost'
is never used \(overwritten on
lines 26 and 30\).](#)

D:\ASU MS IN CS COURSE CONTENT AND
ALL\SEMESTERS COURSES CONTENT\SPRING

20 2023\CSE 565 Software Verification Validation and
Testing\ASSIGNMENTS\Assignment 4 - Decision Code
Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java

22 [Found 'DD'-anomaly for variable
'output' \(lines '22'-'34'\).](#)

| | | |
|---|---|--|
| D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING | | |
| 21 | 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 22 Found 'DD'-anomaly for variable 'output' (lines '22'-'36'). |
| D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING | | |
| 22 | 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 22 The initializer for variable 'output' is never used (overwritten on lines 34 and 36). |
| D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING | | |
| 23 | 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 24 Avoid using Literals in Conditional Statements |
| D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING | | |
| 24 | 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 24 Use equals() to compare object references. |
| D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING | | |
| 25 | 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 24 Use equals() to compare strings instead of '==' or '!=' |
| D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING | | |
| 26 | 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 33 Avoid using Literals in Conditional Statements |
| D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING | | |
| 27 | 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 34 Avoid using if...else statements without curly braces |
| D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING | | |
| 28 | 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 34 This statement should have braces |
| D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING | | |
| 29 | 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 36 Avoid using if...else statements without curly braces |
| D:\ASU MS IN CS COURSE CONTENT AND ALL\SEMESTERS COURSES CONTENT\SPRING | | |
| 30 | 2023\CSE 565 Software Verification Validation and Testing\ASSIGNMENTS\Assignment 4 - Decision Code Coverage\CSE 565 SVVT Assignment 4\src\StaticAnalysis.java | 36 This statement should have braces |