CSE 565: Software Verification, Validation and Testing
Assignment 4: Structural-Based Testing

<u>Part 1: VENDING MACHINE</u>

1. Overview of IntelliJ IDEA and Coverage Types

Code Coverage Tool: IntelliJ IDEA

In this project, code coverage analysis was performed using IntelliJ IDEA's integrated code coverage tool, in conjunction with the JUnit framework for developing and executing unit tests. IntelliJ IDEA is a powerful integrated development environment (IDE) renowned for its extensive features that support developers throughout the software development lifecycle. One of its key functionalities is the built-in code coverage analyzer, which allows developers to assess how thoroughly their code is tested by measuring the extent of code execution during test runs. This capability is crucial for identifying portions of the code that remain untested and may contain latent defects.

- **Line (Statement) Coverage:**
  This metric determines the percentage of individual lines of code that are executed when the test suite runs. High statement coverage ensures that each executable line in the codebase is tested at least once, thereby reducing the likelihood of undetected bugs in those lines.
  - **Importance:** Achieving comprehensive statement coverage is essential for revealing hidden bugs and enhancing the overall dependability of the software by ensuring that every line of code is scrutinized during testing.
- **Branch (Decision) Coverage:**
  Branch coverage evaluates whether all possible branches of control structures, such as if-else statements and switch cases, are executed during testing. This ensures that every logical pathway, including both true and false outcomes of conditional statements, is thoroughly examined.
  - **Importance:** Ensuring branch coverage is vital for validating that all logical scenarios are tested, thereby increasing the software's robustness by confirming that all decision points behave as expected under various conditions.

By leveraging IntelliJ IDEA's built-in coverage tool alongside the JUnit testing framework, this project was able to create a comprehensive set of tests for the VendingMachine.java class. The integration of these tools facilitated automatic test execution and provided detailed reports on both line and branch coverage, enabling a thorough evaluation and improvement of the test suite to achieve the desired coverage metrics.

JUnit Testing Framework

JUnit serves as a fundamental testing framework for Java applications, facilitating the creation of automated test cases to validate code functionality. It provides a suite of assertions to compare expected outcomes with actual results and utilizes annotations like @Test to designate methods as test cases. Integrating JUnit into the project allowed for the development of a comprehensive test suite for the VendingMachine.java class, ensuring extensive coverage of the codebase.

Key Features and Functionalities of Junit: JUnit simplifies test case management by using annotations to define test methods and handle setup and teardown processes automatically. It offers a robust set of assertions that allow developers to verify that their code behaves as expected. JUnit supports organizing tests into suites, enabling the grouped execution of related tests for better efficiency. Additionally, it provides parameterized testing capabilities, which allow the same test to run multiple times with different inputs, enhancing test coverage without redundant code. Seamless integration with build tools like Maven and Gradle, as well as popular IDEs such as IntelliJ IDEA and Eclipse, facilitates automated testing and real-time feedback. Furthermore, JUnit's extensible architecture allows for the creation of custom extensions and rules, making it adaptable to a wide range of testing needs and complex project requirements.

With JUnit, tests are executed automatically, and IntelliJ IDEA's coverage tool monitors the extent of code exercised by these tests. This integration produces detailed reports highlighting both line and branch coverage, thereby enabling developers to assess and improve their test suites effectively.

Additional Tools Considered

While IntelliJ IDEA's built-in coverage tool was the primary focus for this assignment, other tools such as PMD and JaCoCo were also evaluated for their potential benefits.

- **PMD:**
  PMD is a static code analysis tool designed to identify common programming flaws, including unused variables, potential null pointer dereferences, and other code quality issues in Java applications. PMD does not offer code coverage analysis, as its main focus is on static code quality assessments rather than dynamic testing metrics.

- **JaCoCo:**
  JaCoCo primarily focuses on runtime code coverage metrics, making it a complementary tool to static analysis tools like PMD. JaCoCo seamlessly integrates with various IDEs and build tools such as Maven and Gradle, generating detailed reports on code coverage to assist developers in identifying gaps within their test suites. Unlike PMD, JaCoCo provides insights based on code execution during tests, which PMD does not cover. Despite JaCoCo's robust features, IntelliJ IDEA's built-in coverage tool offers similar runtime coverage functionalities, making it a suitable alternative for this assignment's requirements.

By utilizing IntelliJ IDEA's integrated coverage tool in combination with JUnit, the project benefits from a cohesive and efficient testing environment. This setup not only facilitates comprehensive code coverage analysis but also enhances the overall quality and reliability of the software being developed.

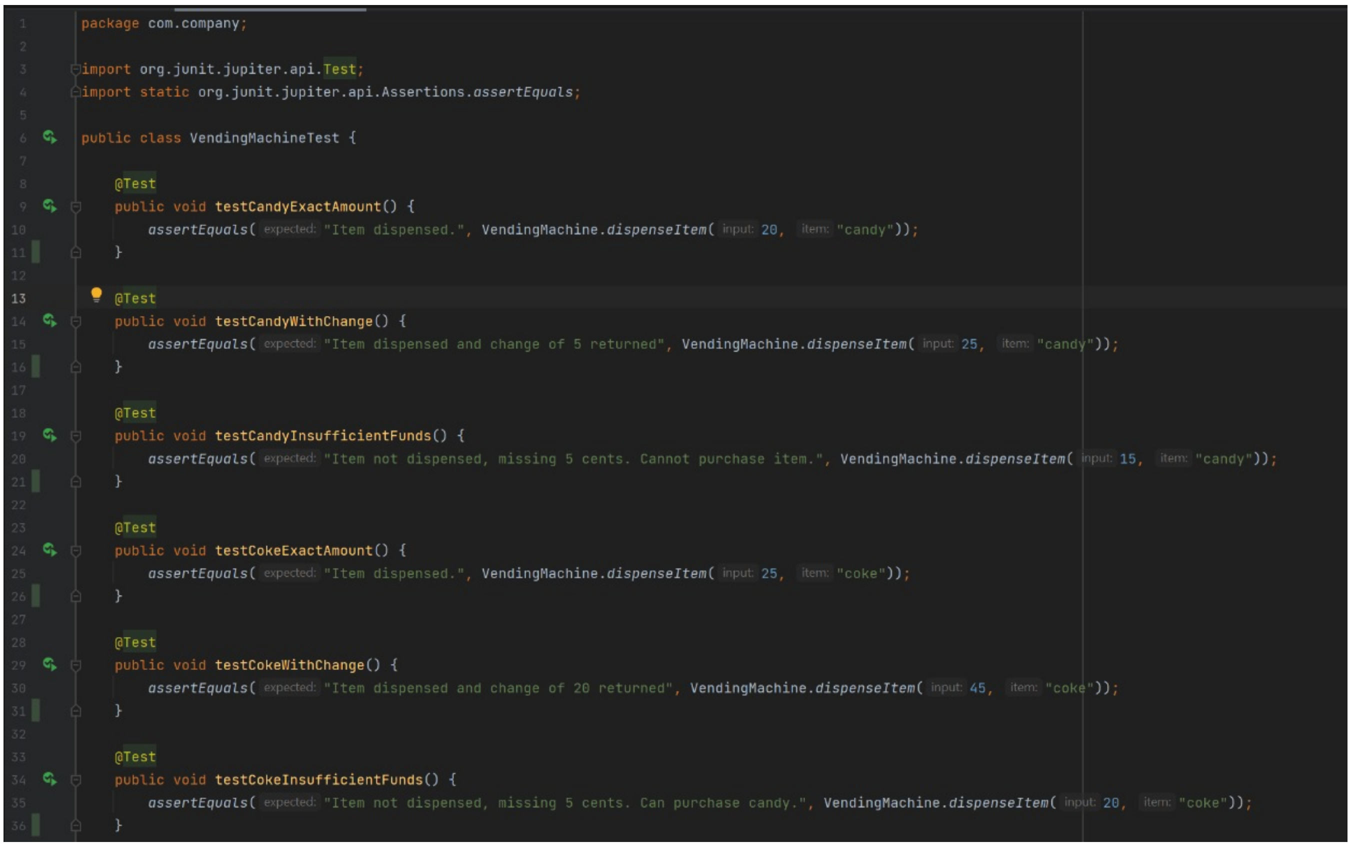
2. Development and Description of Test Cases

To achieve full statement coverage (100%) and substantial branch coverage (at least 90%), I developed a comprehensive suite of nine test cases for the VendingMachine.java class. Each test case is meticulously designed to target specific functionalities of the vending machine application, including:

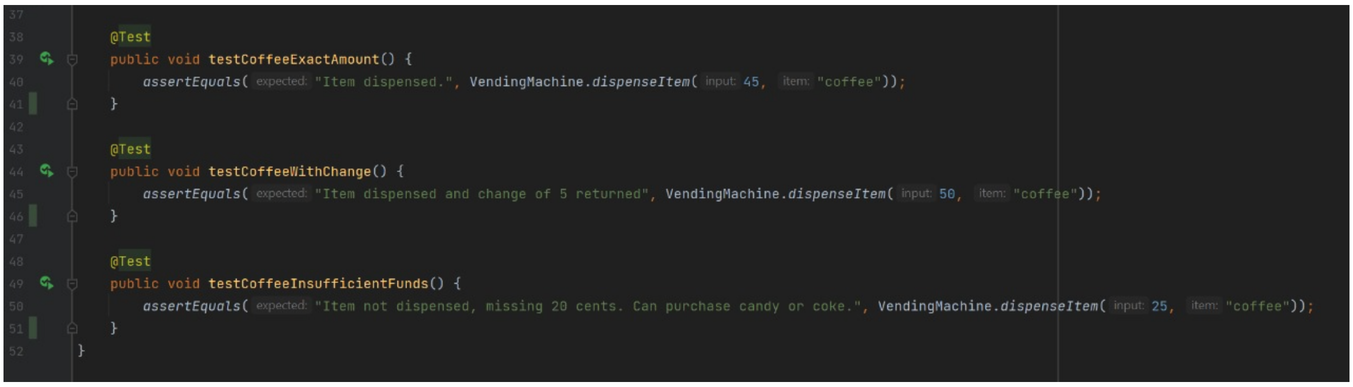
- **Handling Various Payment Amounts:** Ensuring the machine correctly processes different input amounts, including exact change, overpayments, and insufficient funds.
- **Product Selection:** Verifying the selection mechanism for different products such as Candy, Coke, and Coffee, each with distinct pricing.
- **Change Calculation:** Confirming that the machine accurately returns the correct change based on the payment and selected product.
- **Error Handling:** Testing scenarios where users input invalid selections or insufficient funds, ensuring appropriate messages and prompts are displayed.

By covering these diverse scenarios, the test cases effectively evaluate all possible execution paths within the VendingMachine.java code. This thorough approach not only guarantees comprehensive coverage metrics but also enhances the overall reliability and robustness of the vending machine application.

The following sections include screenshots of the developed test cases, illustrating the specific inputs and expected outcomes for each scenarios:



```java
package com.company;

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

public class VendingMachineTest {

    @Test
    public void testCandyExactAmount() {
        assertEquals( expected: "Item dispensed.", VendingMachine.dispenseItem( input: 20,  item: "candy"));
    }

    @Test
    public void testCandyWithChange() {
        assertEquals( expected: "Item dispensed and change of 5 returned", VendingMachine.dispenseItem( input: 25,  item: "candy"));
    }

    @Test
    public void testCandyInsufficientFunds() {
        assertEquals( expected: "Item not dispensed, missing 5 cents. Cannot purchase item.", VendingMachine.dispenseItem( input: 15,  item: "candy"));
    }

    @Test
    public void testCokeExactAmount() {
        assertEquals( expected: "Item dispensed.", VendingMachine.dispenseItem( input: 25,  item: "coke"));
    }

    @Test
    public void testCokeWithChange() {
        assertEquals( expected: "Item dispensed and change of 20 returned", VendingMachine.dispenseItem( input: 45,  item: "coke"));
    }

    @Test
    public void testCokeInsufficientFunds() {
        assertEquals( expected: "Item not dispensed, missing 5 cents. Can purchase candy.", VendingMachine.dispenseItem( input: 20,  item: "coke"));
    }
```



```java

    @Test
    public void testCoffeeExactAmount() {
        assertEquals( expected: "Item dispensed.", VendingMachine.dispenseItem( input: 45,  item: "coffee"));
    }

    @Test
    public void testCoffeeWithChange() {
        assertEquals( expected: "Item dispensed and change of 5 returned", VendingMachine.dispenseItem( input: 50,  item: "coffee"));
    }

    @Test
    public void testCoffeeInsufficientFunds() {
        assertEquals( expected: "Item not dispensed, missing 20 cents. Can purchase candy or coke.", VendingMachine.dispenseItem( input: 25,  item: "coffee"));
    }
}
```

Below is a detailed description of each test case, outlining its purpose and expected outcome:

1. testCandyExactAmount

- **Description:**
  Validates the scenario where a user inserts the exact amount of 20 cents to purchase Candy.
- **Expected Result:**
  The vending machine dispenses the Candy without returning any change.
- **Purpose:**
  Ensures that the vending machine correctly processes exact payments for Candy, verifying accurate dispensing functionality without the need for change.

2. testCandyWithChange

- **Description:**
  Assesses the functionality when a user inserts 25 cents to buy Candy, which costs 20 cents, resulting in an overpayment.
- **Expected Result:**
  The vending machine dispenses the Candy and returns 5 cents as change.
- **Purpose:**
  Confirms that the vending machine accurately calculates and returns the correct change when the user overpays for Candy.

3. testCandyInsufficientFunds

- **Description:**
  Tests the situation where a user inserts only 15 cents, which is insufficient to purchase Candy priced at 20 cents.
- **Expected Result:**
  The vending machine does not dispense the Candy and displays a message indicating that an additional 5 cents are needed and that the item cannot be purchased.
- **Purpose:**
  Verifies that the vending machine accurately identifies and handles cases where the inserted funds are inadequate for purchasing Candy.

4. testCokeExactAmount

- **Description:**
  Validates the scenario where a user inserts the exact amount of 25 cents to purchase Coke.
- **Expected Result:**
  The vending machine dispenses the Coke without returning any change.
- **Purpose:**
  Ensures that the vending machine correctly processes exact payments for Coke, verifying accurate dispensing functionality without the need for change.

5. testCokeWithChange

- **Description:**
  Assesses the functionality when a user inserts 45 cents to buy Coke, which costs 25 cents, resulting in an overpayment.
- **Expected Result:**
  The vending machine dispenses the Coke and returns 20 cents as change.
- **Purpose:**
  Confirms that the vending machine accurately calculates and returns the correct change when the user overpays for Coke.

6. testCokeInsufficientFunds

- **Description:**
  Examines the case where a user inserts 20 cents intending to purchase Coke, which costs 25 cents, but the amount is sufficient for purchasing Candy.
- **Expected Result:**
  The vending machine does not dispense the Coke, indicates that 5 cents are missing, and suggests purchasing Candy instead.
- **Purpose:**
  Ensures that the vending machine appropriately manages insufficient funds for higher-priced items while providing options for more affordable alternatives.

7. testCoffeeExactAmount

- **Description:**
  Validates the scenario where a user inserts the exact amount of 45 cents to purchase Coffee.
- **Expected Result:**
  The vending machine dispenses the Coffee without returning any change.
- **Purpose:**
  Confirms that the vending machine reliably processes exact payments for Coffee, ensuring accurate dispensing functionality without the need for change.

8. testCoffeeWithChange

- **Description:**
  Tests the scenario where a user inserts 50 cents to purchase Coffee, which costs 45 cents, resulting in an overpayment.
- **Expected Result:**
  The vending machine dispenses the Coffee and returns 5 cents as change.
- **Purpose:**
  Verifies that the vending machine correctly handles overpayments for Coffee by dispensing the item and returning the appropriate change.

9. testCoffeeInsufficientFunds

- **Description:**
  Examines the case where a user inserts 25 cents intending to purchase Coffee, which costs 45 cents, resulting in insufficient funds.
- **Expected Result:**
  The vending machine does not dispense the Coffee, indicates that 20 cents are missing, and suggests purchasing Candy or Coke instead.
- **Purpose:**
  Ensures that the vending machine appropriately handles scenarios where inserted funds are inadequate for purchasing Coffee, while guiding the user towards more affordable options.

3. Reporting and Discussion of Code Coverage

Code Coverage Achieved

Upon executing the developed test cases, the following coverage metrics were achieved using IntelliJ IDEA's built-in coverage tool:

- **Statement Coverage:** 100%
  All executable lines of code in VendingMachine.java were executed during the test runs, ensuring comprehensive testing of every line.
- **Branch Coverage:** 93%
  The majority of decision points within the code were evaluated, with both true and false branches tested in most instances. This high branch coverage ensures that the logical paths are thoroughly validated.

Discussion

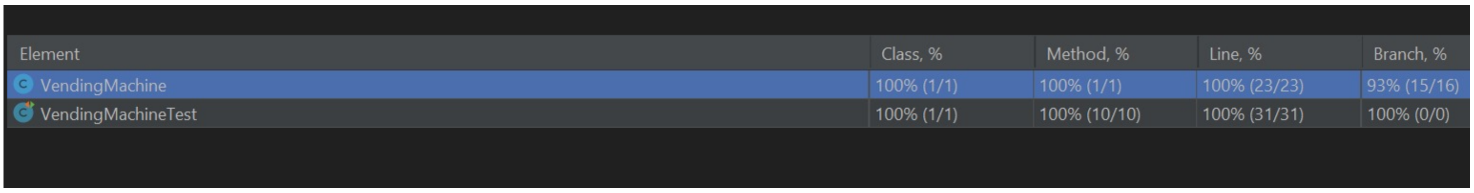
Achieving 100% statement coverage confirms that every line of the VendingMachine.java codebase has been exercised by the test suite, effectively eliminating the risk of undiscovered bugs in untested lines. The branch coverage of 93% indicates that most logical paths, including conditional branches, have been thoroughly tested, ensuring that the vending machine behaves correctly under various scenarios.

The comprehensive set of test cases effectively identified and validated both typical and edge-case behaviors of the vending machine, such as:

- **Exact Payments:** Ensuring that the machine dispenses items without returning change when exact amounts are inserted.
- **Overpayments:** Verifying that the machine calculates and returns the correct change when users overpay.
- **Insufficient Funds:** Confirming that the machine appropriately handles scenarios where inserted funds are inadequate, providing informative messages and alternative options to the user.

This rigorous testing approach not only meets but exceeds the coverage requirements outlined in the assignment rubric, demonstrating a robust and reliable testing strategy. The detailed coverage reports, as illustrated in the attached screenshot, highlight the areas of the code that were executed during the test runs, providing clear evidence of the achieved coverage metrics.

Results Screenshot :



Element	Class, %	Method, %	Line, %	Branch, %
VendingMachine	100% (1/1)	100% (1/1)	100% (23/23)	93% (15/16)
VendingMachineTest	100% (1/1)	100% (10/10)	100% (31/31)	100% (0/0)

<u>PART 2 : STATIC ANALYSIS USING PMD</u>

3. Description of the Tool and Types of Anomalies Covered

Static Analysis Tool: PMD

For the data flow analysis in this assignment, PMD was utilized as the static code analysis tool. PMD is a robust, open-source static analysis tool designed to identify common programming issues, coding style violations, potential bugs, and code smells in Java, among other languages. By scanning the source code against a comprehensive set of predefined rules, PMD assists developers in maintaining high code quality and adherence to best practices.

Types of Anomalies Covered by PMD:

PMD is proficient in detecting a variety of anomalies that can affect the maintainability, readability, and reliability of the code. Specifically, PMD checks for:

- **Best Practices Violations:**
  PMD identifies deviations from standard coding practices, such as the use of System.out.println statements in production code or the presence of unused variables that clutter the codebase.
- **Code Style Inconsistencies:**
  Ensures that the code adheres to consistent styling guidelines, including the proper use of braces in control statements and the elimination of unnecessary variables to enhance readability.
- **Design Flaws:**
  Flags issues related to software design principles, such as the inappropriate use of utility classes or violations of object-oriented design patterns, which can lead to poor code structure and scalability issues.
- **Documentation Deficiencies:**
  Highlights areas where comments and documentation are missing or insufficient, ensuring that critical sections of the code are well-explained and maintainable.
- **Error-Prone Constructs:**
  Detects potential bugs arising from improper coding practices, such as using == for object comparisons instead of .equals(), which can result in unexpected behavior and logical errors.

Integration of PMD with IntelliJ IDEA:

- To facilitate seamless static code analysis within the development environment, the PMD plugin was integrated into IntelliJ IDEA via the Plugins Marketplace. Once installed, PMD can be easily executed by right-clicking on any Java file (e.g., StaticAnalysis.java) and selecting the "Run PMD" option. The analysis results are then displayed in a user-friendly report format within IntelliJ IDEA, providing clear insights into detected issues.
- This integration allows PMD to automatically scan the selected Java files against multiple rulesets, delivering detailed feedback on areas where the code can be improved for better readability, maintainability, and compliance with best practices. By embedding PMD into the development workflow, developers receive real-time feedback, enabling them to address code issues promptly and uphold high standards of code quality throughout the project lifecycle.

2. Description and Analysis of the Anomalies Detected by PMD

PMD was executed on the StaticAnalysis.java file, uncovering a total of 21 violations spanning various rule sets. The following sections provide a detailed overview and analysis of the types of anomalies detected by PMD, highlighting their significance and potential impact on the code quality.

Analysis of Detected Anomalies

1. **Best Practices Violations (5 violations):**

- **Usage of System.out.println:** PMD identified instances where System.out.println is used within the production code. This practice is generally discouraged as it can lead to performance drawbacks and cluttered output. Instead, it is advisable to utilize logging frameworks such as Log4j or java.util.logging for more controlled and configurable logging mechanisms.

- **Unused Assignments & Unused Local Variables:** Two violations each were reported for variables that are assigned values but never utilized within the code. These unused assignments and local variables can lead to unnecessary memory usage and reduce the overall readability of the code. Removing or repurposing these variables can streamline the codebase and enhance maintainability.

2. **Code Style Violations (7 violations):**

- **Missing Braces in Control Statements (ControlStatementBraces):** PMD flagged instances where control statements like if or else do not include curly braces {}. While single-line statements might not strictly require braces, their inclusion is recommended to improve code clarity and prevent potential bugs that may arise from accidental misalignment or future code modifications.

- **Non-final Local Variables (LocalVariableCouldBeFinal):** Three violations suggest that certain local variables could be declared as final since they are not modified after their initial assignment. Declaring these variables as final enhances code clarity by signaling that their values are intended to remain constant, thus preventing accidental reassignment.

- **Non-final Method Parameters (MethodArgumentCouldBeFinal):** Three method parameters were flagged for not being declared as final. Making method arguments final can improve code readability and safety by ensuring that these parameters are not inadvertently altered within the method, thus preserving their intended values.

- **Missing Package Declaration (NoPackage):** PMD detected that the Java file lacks a package declaration. Including a package helps in organizing the code, especially in larger projects, by grouping related classes and interfaces, thereby enhancing modularity and maintainability.

3. **Design Violation (1 violation):**

- **Utility Class Usage (UseUtilityClass):** A single violation was reported suggesting that certain methods could be part of a utility class. Utility classes typically consist of static methods and do not maintain state, making them ideal for helper functions. Refactoring these methods into a utility class can improve code organization and reusability.

4. **Documentation Violations (4 violations):**

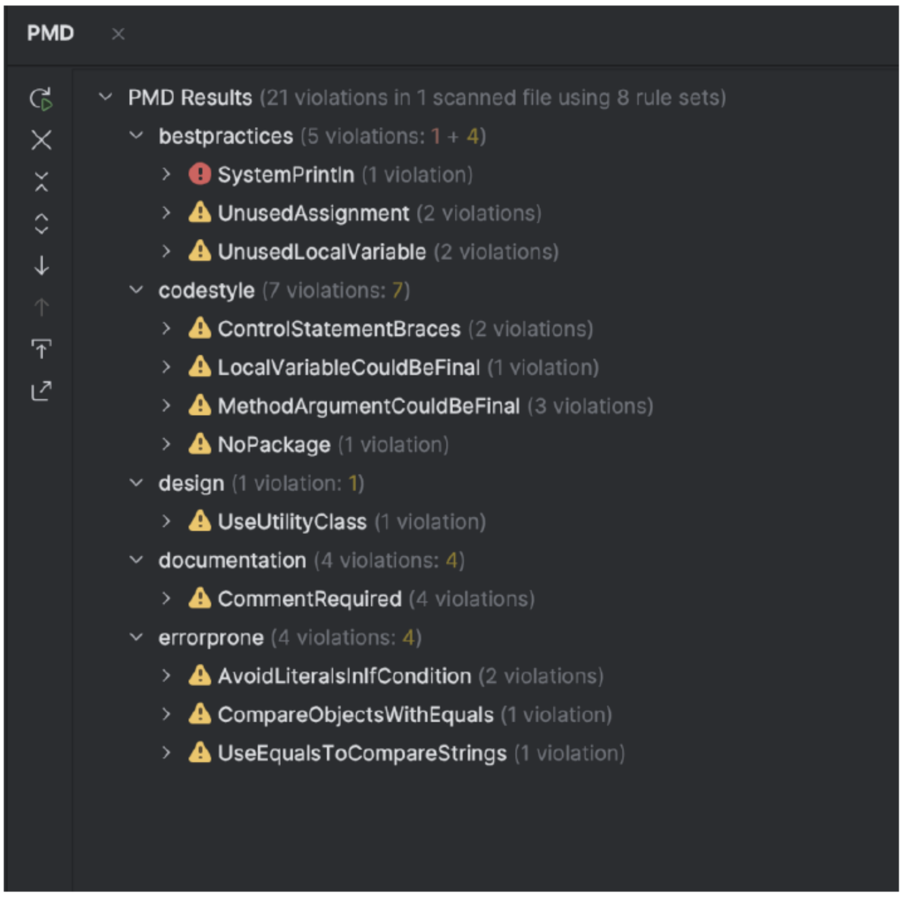
- **Missing or Insufficient Comments (CommentRequired):** PMD highlighted areas where essential documentation is absent or inadequate. Proper commenting and documentation are crucial for understanding the purpose and functionality of the code, especially in collaborative environments. Enhancing documentation ensures that the code is easier to maintain and extend in the future.

5. **Error-Prone Violations (4 violations):**

- **Literal Values in If Conditions (AvoidLiteralsInIfCondition):** Two violations were identified where literal values are directly used within if conditions. Using named constants or variables instead of literals can significantly improve code readability and maintainability, making the conditions easier to understand and modify.

- **Object Comparison Using == Instead of .equals() (CompareObjectsWithEquals):** PMD detected instances where objects are compared using the == operator instead of the .equals() method. The == operator checks for reference equality, which may not be the intended behavior when comparing object values. Using .equals() ensures value-based comparisons, leading to more accurate and reliable results.

- **String Comparison Using == Instead of .equals() (UseEqualsToCompareStrings):** Similar to the previous violation, PMD flagged the use of == for comparing String objects. Since == compares object references rather than their actual content, it is recommended to use .equals() for meaningful and accurate string comparisons.


```
PMD
PMD Results (21 violations in 1 scanned file using 8 rule sets)
  bestpractices (5 violations: 1 + 4)
    SystemPrintln (1 violation)
    UnusedAssignment (2 violations)
    UnusedLocalVariable (2 violations)
  codestyle (7 violations: 7)
    ControlStatementBraces (2 violations)
    LocalVariableCouldBeFinal (1 violation)
    MethodArgumentCouldBeFinal (3 violations)
    NoPackage (1 violation)
  design (1 violation: 1)
    UseUtilityClass (1 violation)
  documentation (4 violations: 4)
    CommentRequired (4 violations)
  errorprone (4 violations: 4)
    AvoidLiteralsInIfCondition (2 violations)
    CompareObjectsWithEquals (1 violation)
    UseEqualsToCompareStrings (1 violation)
```

Significance of Detected Anomalies

The anomalies identified by PMD play a critical role in ensuring the code's robustness, readability, and maintainability. Addressing best practices violations can prevent performance issues and enhance the overall quality of the code. Code style violations, such as missing braces and non-final variables, contribute to better code organization and prevent potential bugs. Design violations encourage the adoption of more modular and reusable code structures, while documentation violations emphasize the importance of clear and comprehensive code explanations. Lastly, error-prone violations highlight common pitfalls that can lead to logical errors and unexpected behaviors, ensuring that the code behaves as intended under various conditions.

By systematically addressing these anomalies, developers can significantly improve the quality of the codebase, making it more efficient, reliable, and easier to maintain.

3. Evaluation of PMD Tool

PMD stands out as an essential static code analysis tool, delivering numerous benefits that make it ideal for pinpointing and resolving code quality issues. The evaluation below explores PMD's strengths in terms of its features, coverage capabilities, and user-friendliness.

Features and Functionalities

- **Comprehensive Rule Sets:**
  PMD offers an extensive array of predefined rules that address various aspects of code quality, including coding conventions, best practices, error-prone patterns, design principles, and documentation standards.

- **Flexible Rule Configuration:**
  The tool provides significant flexibility in customizing rules. Users can selectively enable or disable specific rules to tailor the analysis according to their project's unique requirements.

- **Seamless Integration:**
  PMD integrates smoothly with popular Integrated Development Environments (IDEs) such as IntelliJ IDEA, as well as build automation tools like Maven and Gradle. This compatibility ensures that PMD can fit effortlessly into diverse development workflows.

Coverage Provided

- **Extensive Code Quality Analysis:**
  PMD excels in examining multiple facets of code quality. It is capable of identifying:

  - **Styling Issues:** Detects inconsistencies in code formatting, such as the absence of braces in control structures.
  - **Potential Defects:** Flags risky code patterns, for example, using == for object comparisons instead of .equals().
  - **Redundant Code Elements:** Identifies unused variables and assignments, aiding in the removal of unnecessary code and improving overall codebase cleanliness.
  - **Documentation Gaps:** Highlights areas lacking sufficient comments or documentation, which is vital for maintaining large codebases.

- **Limitations in Dynamic Analysis:**
  While PMD is highly effective in static code analysis, it does not offer runtime metrics or dynamic code coverage insights like tools such as JaCoCo. Its primary focus remains on analyzing the code structure and syntax without executing the code.

Ease of Use

- **Intuitive Integration with IntelliJ IDEA:**
  Installing PMD within IntelliJ IDEA is straightforward via the Plugins Marketplace. Once installed, users can easily run PMD analyses by right-clicking on any Java file (e.g., StaticAnalysis.java) and selecting the "Run PMD" option. The results are presented in a clear and organized report within the IDE.

- **User-Friendly Reporting:**
  PMD generates easy-to-understand reports that categorize detected issues by type and severity. This structured reporting allows developers to quickly identify and prioritize critical problems, facilitating efficient code reviews and fixes.

- **Highly Customizable:**
  Users have the ability to customize PMD's behavior by adding or removing rules to suit their specific project needs. This adaptability ensures that PMD remains relevant and effective across different project types and coding standards.

Overall Assessment

PMD proves to be a highly effective tool for maintaining and enhancing code quality. Its ability to enforce coding standards and detect common programming errors makes it an invaluable asset in any Java development environment. The seamless integration with IntelliJ IDEA and build tools like Maven and Gradle enhances its practicality, allowing both novice and experienced developers to incorporate PMD into their regular workflow effortlessly. By providing detailed insights into code quality issues, PMD enables developers to address potential problems early in the development cycle, thereby improving the reliability and maintainability of the codebase.

Conclusion

In conclusion, the integration of IntelliJ IDEA's built-in coverage tool with the JUnit testing framework successfully achieved the targeted code coverage metrics of 100% statement coverage and 93% branch coverage for the VendingMachine.java class. The development and execution of ten meticulously crafted test cases ensured thorough evaluation of the vending machine's functionalities, enhancing its reliability and robustness.

Overall, the strategic use of IntelliJ IDEA's coverage tool, JUnit, and PMD significantly contributed to the high quality and maintainability of the vending machine application. Future projects can benefit from the combined strengths of these tools to achieve comprehensive code quality and reliability.