Structural-Based Testing Assignment

Name – Prasidh Aggarwal
ASURITE – 1225362125
Email – paggar10@asu.edu

Part 1

Tool Used - JaCoCo

Description and Reasons:

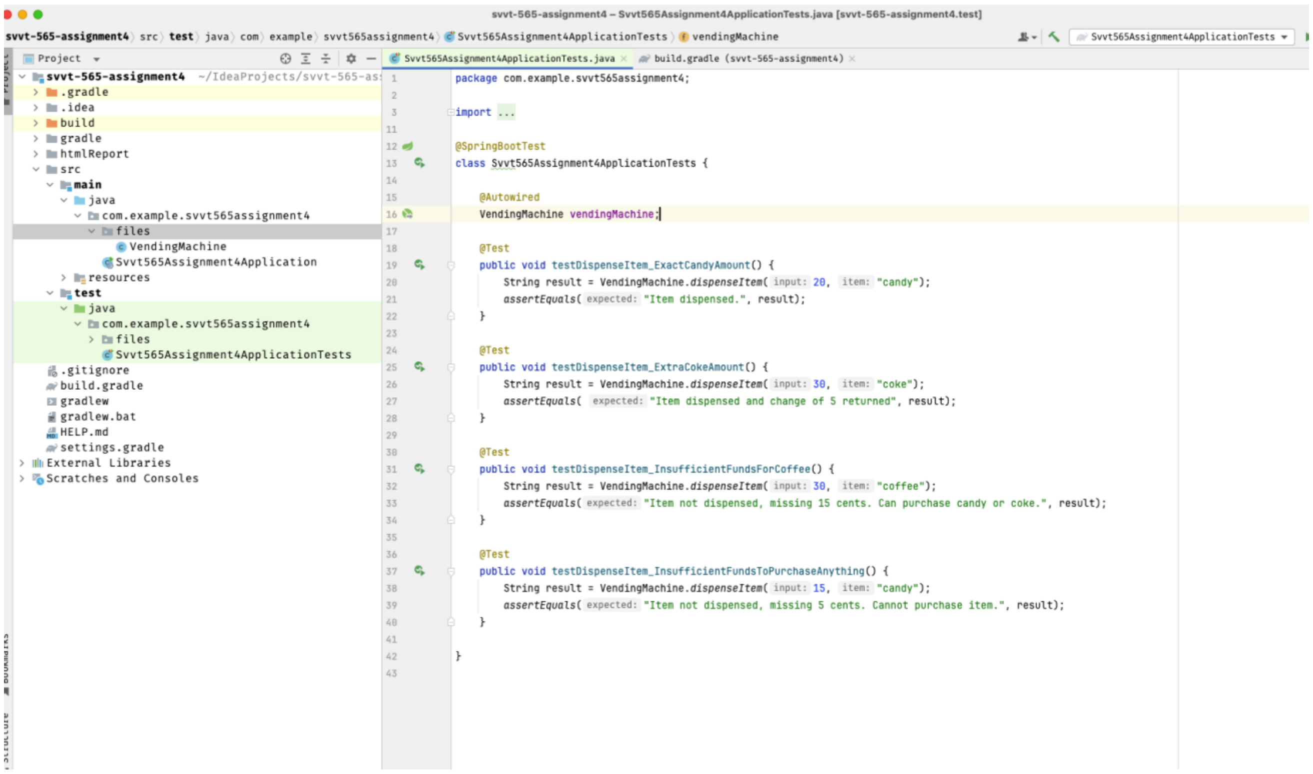
1. **Ease of Integration**: JaCoCo seamlessly integrates with popular build tools like Maven and Gradle, making it easy to incorporate into your existing development workflow without significant configuration overhead. It also supports various CI/CD platforms, ensuring smooth integration into your automated build and test pipelines.
2. **Comprehensive Coverage Metrics**: JaCoCo provides comprehensive coverage metrics, including line coverage, branch coverage, instruction coverage, and more. These metrics offer detailed insights into the quality and completeness of your test suite, helping you identify areas of code that require additional testing.
3. **Lightweight and Fast**: JaCoCo is known for its lightweight and fast instrumentation process, which minimally impacts the performance of your applications during testing. This efficiency ensures that you can achieve accurate code coverage results without sacrificing the speed of your test execution.
4. **Wide Adoption and Community Support**: JaCoCo has gained widespread adoption within the Java community, with many projects and organizations leveraging its capabilities for code coverage analysis. This widespread usage translates to extensive community support, including documentation, tutorials, and troubleshooting resources, making it easier to get help when needed.
5. **Open Source and Active Development**: JaCoCo is an open-source project with active development and maintenance. This means that it is continuously evolving to meet the needs of modern software development practices and to address emerging challenges in code coverage analysis. Additionally, being open-source fosters transparency and allows users to contribute improvements and fixes to the tool.

Code coverage JaCoCo provides:

1. **Line Coverage**: Measures the percentage of code lines executed by tests.
2. **Branch Coverage**: Determines the percentage of decision points (like if statements) covered by tests.

3. **Instruction Coverage**: Measures the percentage of bytecode instructions executed by tests.
4. **Method Coverage**: Calculates the percentage of methods invoked during testing.
5. **Class Coverage**: Evaluates the percentage of classes instantiated or loaded by tests.
6. **Complexity Coverage**: Assesses code complexity, including metrics like cyclomatic complexity.

Test cases developed:



svvt-565-assignment4 – Svvt565Assignment4ApplicationTests.java [svvt-565-assignment4.test]

```java
package com.example.svvt565assignment4;

import ...

@SpringBootTest
class Svvt565Assignment4ApplicationTests {

    @Autowired
    VendingMachine vendingMachine;

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

    @Test
    public void testDispenseItem_ExtraCokeAmount() {
        String result = VendingMachine.dispenseItem( input: 30,  item: "coke");
        assertEquals(  expected: "Item dispensed and change of 5 returned", result);
    }

    @Test
    public void testDispenseItem_InsufficientFundsForCoffee() {
        String result = VendingMachine.dispenseItem( input: 30,  item: "coffee");
        assertEquals( expected: "Item not dispensed, missing 15 cents. Can purchase candy or coke.", result);
    }

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

}
```

Description of each test case:

1. **testDispenseItem_ExactCandyAmount**:
   - This test case verifies that when the user inputs exactly the amount required for purchasing candy (20 cents), the vending machine dispenses the candy without any change returned.
   - It ensures that the method correctly handles the scenario where the input amount matches the cost of the item.

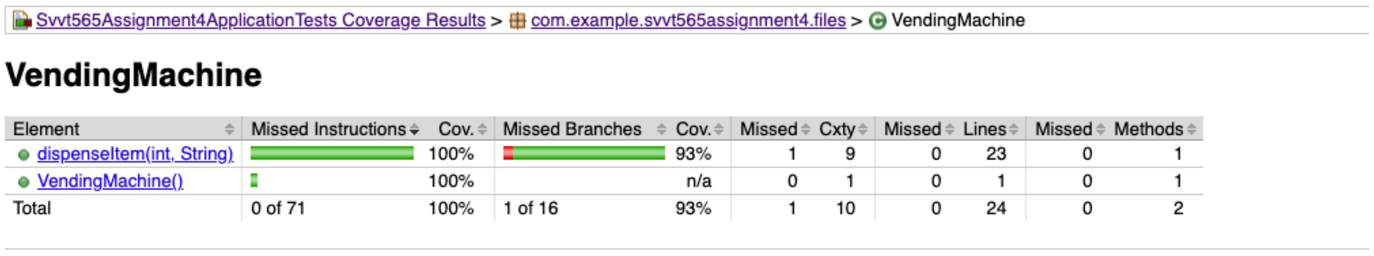
2. **testDispenseItem_ExtraCokeAmount**:
   - This test case validates that when the user inputs more than the required amount for purchasing coke (30 cents), the vending machine dispenses the coke and returns the correct amount of change (5 cents).

- It checks whether the method correctly calculates and returns the change when the input amount exceeds the cost of the item.

3. **testDispenseItem_InsufficientFundsForCoffee**:
   - This test case ensures that when the user inputs an amount insufficient to purchase coffee (30 cents), the vending machine provides an appropriate message indicating the remaining amount needed to purchase coffee or other available items.
   - It verifies the method's behavior when the input amount is insufficient to purchase the specified item, testing the generation of the correct error message.

4. **testDispenseItem_InsufficientFundsToPurchaseAnything**:
   - This test case verifies the vending machine's response when the user inputs an amount insufficient to purchase any item (15 cents). It expects the vending machine to inform the user that the item cannot be purchased due to insufficient funds.
   - It checks whether the method correctly handles scenarios where the input amount is insufficient to purchase any available item, testing the generation of the appropriate error message.

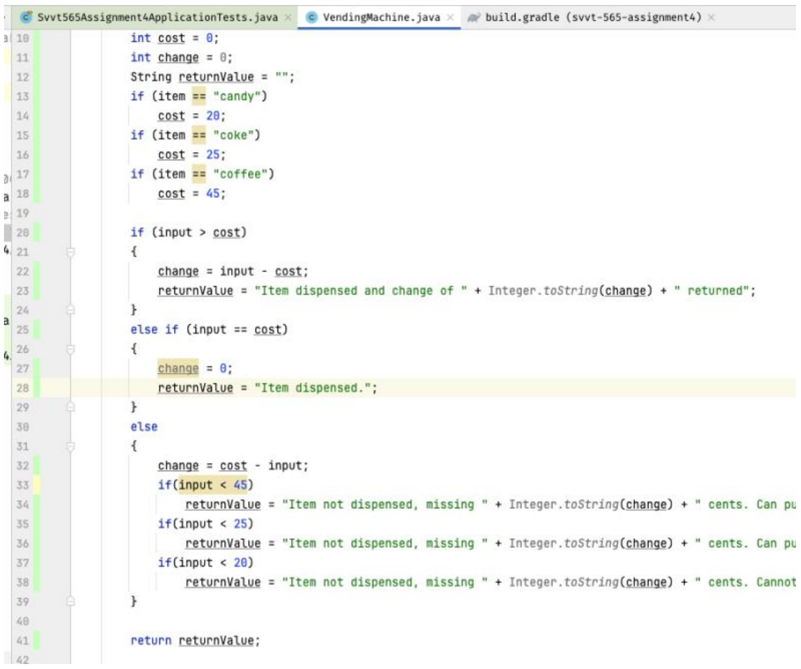
JaCoCo Coverage Report:



Svvt565Assignment4ApplicationTests Coverage Results > com.example.svvt565assignment4.files > VendingMachine

VendingMachine

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
dispenseItem(int, String)		100%		93%	1	9	0	23	0	1
VendingMachine()		100%		n/a	0	1	0	1	0	1
Total	0 of 71	100%	1 of 16	93%	1	10	0	24	0	2



Svvt565Assignment4ApplicationTests.java | VendingMachine.java | build.gradle (svvt-565-assignment4)

```java
        int cost = 0;
        int change = 0;
        String returnValue = "";
        if (item == "candy")
            cost = 20;
        if (item == "coke")
            cost = 25;
        if (item == "coffee")
            cost = 45;

        if (input > cost)
        {
            change = input - cost;
            returnValue = "Item dispensed and change of " + Integer.toString(change) + " returned";
        }
        else if (input == cost)
        {
            change = 0;
            returnValue = "Item dispensed.";
        }
        else
        {
            change = cost - input;
            if(input < 45)
                returnValue = "Item not dispensed, missing " + Integer.toString(change) + " cents. Can pu
            if(input < 25)
                returnValue = "Item not dispensed, missing " + Integer.toString(change) + " cents. Can pu
            if(input < 20)
                returnValue = "Item not dispensed, missing " + Integer.toString(change) + " cents. Cannot
        }

        return returnValue;
```

Results evaluated from the JaCoCo Report:

- Statement Coverage: All the instructions/statements were covered, hence achieving 100% statement coverage.
- Branch/Decision Coverage: The branch with input < 45 is missed because it's a bug in the code. That statement should ideally be placed last because anything less than 20 or 25 will obviously be less than 45. Hence, that branch is never actually implemented. Therefore, the branch coverage achieved is 93%.
- Line Coverage: Line coverage achieved is 100% since we can reach every line in the code.
- Method Coverage: There is only one method named dispenseItem, and its covered. Also, since we are using auto-wiring, the default class constructor for Vending machine is covered as well.

Part 2

Tool Used – PMD

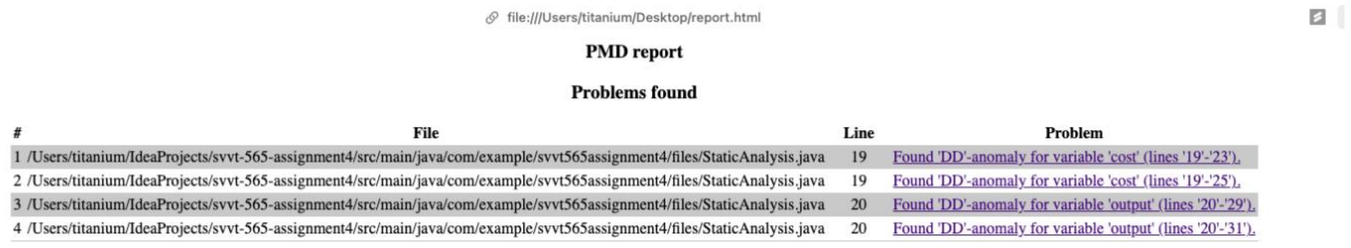
Why PMD?

PMD stands out as an apt tool for static analysis due to its comprehensive language support, customizable rule sets, and ease of integration into existing development workflows. Its capability to pinpoint a wide array of issues—from code style inconsistencies and potential bugs to security vulnerabilities and performance bottlenecks—makes it incredibly valuable for maintaining high-quality codebases. Furthermore, PMD's focus on not just detecting problems but also suggesting improvements aligns with the proactive approach needed in modern software development. By providing actionable insights, PMD empowers developers to address issues early in the development cycle, leading to more reliable and maintainable software projects.

Data Flow Anomalies Identified:

- Found DD anomaly for variable cost
- Found DD anomaly for variable output

Screenshots of Data flow Anomalies:

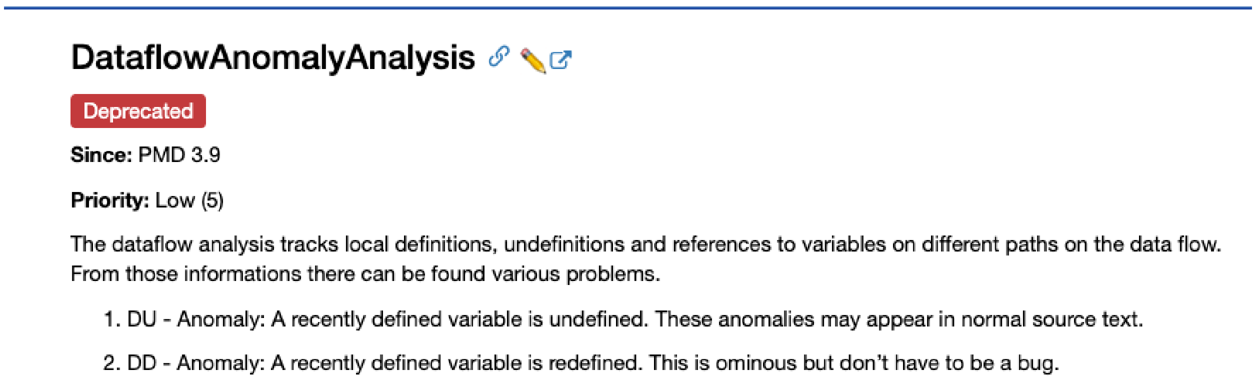


file:///Users/titanium/Desktop/report.html

PMD report

Problems found

#	File	Line	Problem
1	/Users/titanium/IdeaProjects/svvt-565-assignment4/src/main/java/com/example/svvt565assignment4/files/StaticAnalysis.java	19	Found 'DD'-anomaly for variable 'cost' (lines '19'-'23').
2	/Users/titanium/IdeaProjects/svvt-565-assignment4/src/main/java/com/example/svvt565assignment4/files/StaticAnalysis.java	19	Found 'DD'-anomaly for variable 'cost' (lines '19'-'25').
3	/Users/titanium/IdeaProjects/svvt-565-assignment4/src/main/java/com/example/svvt565assignment4/files/StaticAnalysis.java	20	Found 'DD'-anomaly for variable 'output' (lines '20'-'29').
4	/Users/titanium/IdeaProjects/svvt-565-assignment4/src/main/java/com/example/svvt565assignment4/files/StaticAnalysis.java	20	Found 'DD'-anomaly for variable 'output' (lines '20'-'31').



DataflowAnomalyAnalysis

Deprecated

Since: PMD 3.9

Priority: Low (5)

The dataflow analysis tracks local definitions, undefinitions and references to variables on different paths on the data flow. From those informations there can be found various problems.

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

Analysis of other code smells detected by PMD:

1. Replace the use of System.out.println() with a logger. Documentation of the error:

   ''References to System.(out|err).print are usually intended for debugging purposes and can remain in the codebase even in production code. By using a logger one can enable/disable this behaviour at will (and by priority) and avoid clogging the Standard out log.''

2. Avoiding the use of '==' and instead using the equals() method. Documentation of the error:

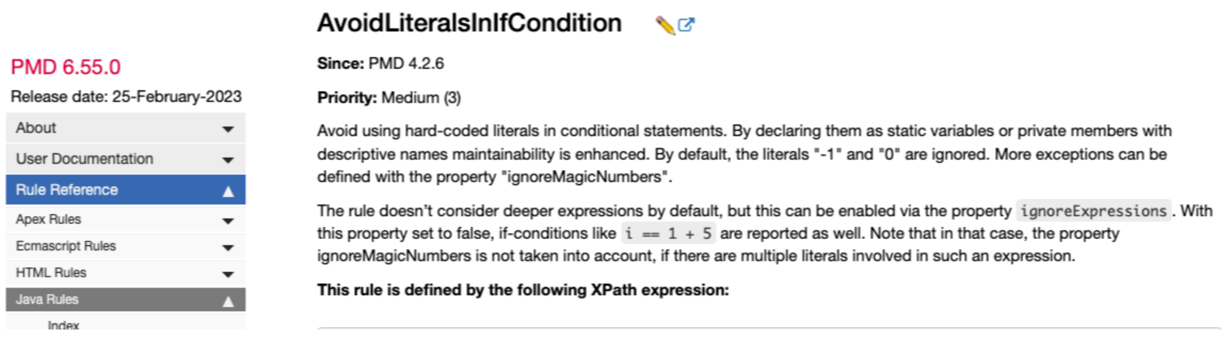
   Using '==' or '!=' to compare strings only works if intern version is used on both sides.

Use the equals() method instead.

3. Many variables in the code can directly be set as final. Documentation of the code smell:

   A method argument that is never re-assigned within the method can be declared final.

4. Avoiding Literals in the If conditions:



AvoidLiteralsInIfCondition

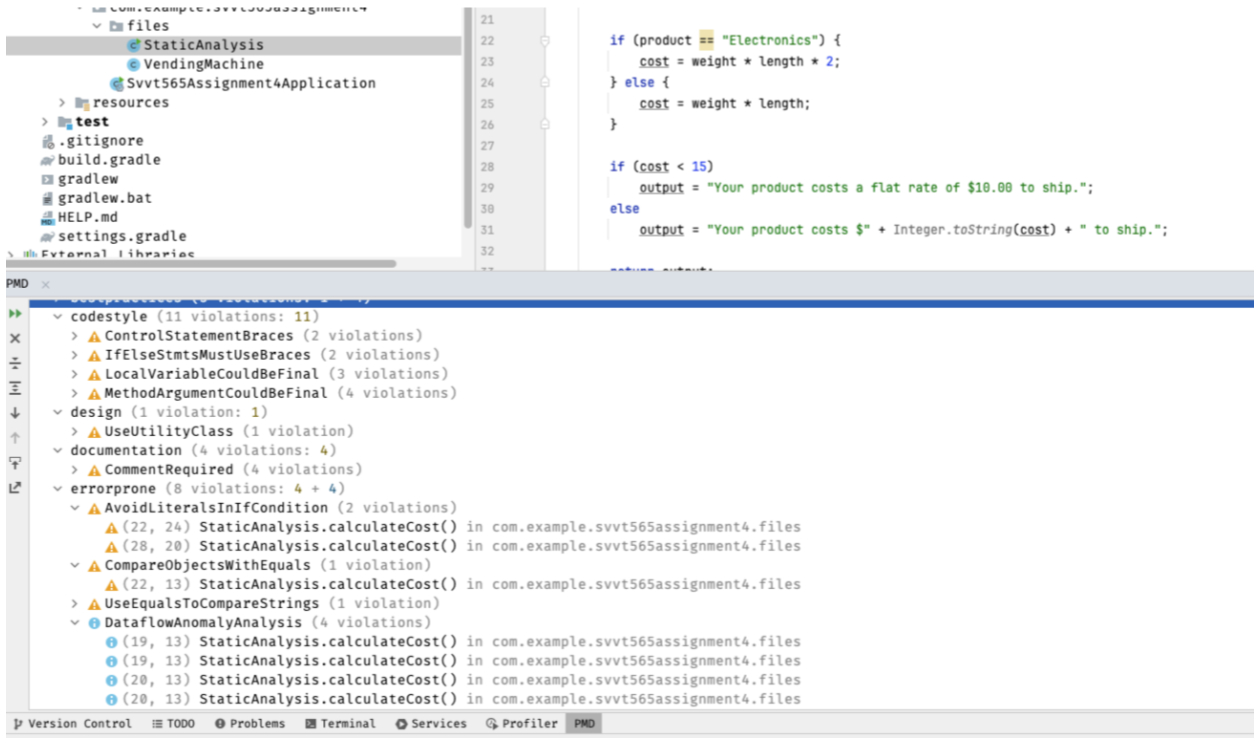
Since: PMD 4.2.6

Priority: Medium (3)

Avoid using hard-coded literals in conditional statements. By declaring them as static variables or private members with descriptive names maintainability is enhanced. By default, the literals "-1" and "0" are ignored. More exceptions can be defined with the property "ignoreMagicNumbers".

The rule doesn't consider deeper expressions by default, but this can be enabled via the property `ignoreExpressions`. With this property set to false, if-conditions like `i == 1 + 5` are reported as well. Note that in that case, the property ignoreMagicNumbers is not taken into account, if there are multiple literals involved in such an expression.

This rule is defined by the following XPath expression:



```
21
22      if (product == "Electronics") {
23          cost = weight * length * 2;
24      } else {
25          cost = weight * length;
26      }
27
28      if (cost < 15)
29          output = "Your product costs a flat rate of $10.00 to ship.";
30      else
31          output = "Your product costs $" + Integer.toString(cost) + " to ship.";
32
```

```
codestyle (11 violations: 11)
  ControlStatementBraces (2 violations)
  IfElseStmtsMustUseBraces (2 violations)
  LocalVariableCouldBeFinal (3 violations)
  MethodArgumentCouldBeFinal (4 violations)
design (1 violation: 1)
  UseUtilityClass (1 violation)
documentation (4 violations: 4)
  CommentRequired (4 violations)
errorprone (8 violations: 4 + 4)
  AvoidLiteralsInIfCondition (2 violations)
    (22, 24) StaticAnalysis.calculateCost() in com.example.svvt565assignment4.files
    (28, 20) StaticAnalysis.calculateCost() in com.example.svvt565assignment4.files
  CompareObjectsWithEquals (1 violation)
    (22, 13) StaticAnalysis.calculateCost() in com.example.svvt565assignment4.files
  UseEqualsToCompareStrings (1 violation)
  DataflowAnomalyAnalysis (4 violations)
    (19, 13) StaticAnalysis.calculateCost() in com.example.svvt565assignment4.files
    (19, 13) StaticAnalysis.calculateCost() in com.example.svvt565assignment4.files
    (20, 13) StaticAnalysis.calculateCost() in com.example.svvt565assignment4.files
    (20, 13) StaticAnalysis.calculateCost() in com.example.svvt565assignment4.files
```

Analysis of PMD (Static analysis tool):

- **Features and Functionalities**
    - **Code Analysis:** The plugin performs static code analysis on Java source code to detect potential issues, such as coding style violations, performance inefficiencies, and potential bugs.
    - **Customizable Rulesets:** PMD offers a wide range of rules to analyze Java code. The plugin allows users to customize which rules are applied to their projects, tailoring the analysis to suit specific coding standards or project requirements.
    - **On-the-fly Feedback:** The PMD plugin provides real-time feedback within the IntelliJ IDEA editor, highlighting code segments that violate PMD rules directly in the source code. This immediate feedback helps developers identify and address issues as they write or modify code.
    - **Integration with Inspection Results:** The plugin integrates PMD analysis results with IntelliJ IDEA's inspection results window, providing a consolidated view of all code quality issues detected by various analysis tools.
    - **Quick Fixes and Refactoring Suggestions:** In addition to highlighting issues, the plugin often provides quick fixes and refactoring suggestions to help developers resolve detected problems efficiently.

- **Coverage Provided**
    - **Code Style**: Covers coding conventions and styles, ensuring that the codebase follows consistent practices for readability and maintainability. It includes checks for naming conventions, code block structures, unnecessary complexity, and size violations.
    - **Error Prone**: Identifies potential bugs that are often the result of typos, misunderstandings of APIs, or misuse of language features. Examples include unnecessary object creation, potential null pointer dereferences, and misuse of APIs.
    - **Security**: Focuses on identifying vulnerabilities or insecure coding practices that could lead to security breaches. This includes hard-coded credentials, SQL injection risks, and other security-related issues.
    - **Performance**: Highlights areas of the code that may be inefficient and could be optimized for better performance. This involves detecting unnecessary object creation, suboptimal usage of APIs, and other performance-related concerns.
    - **Best Practices**: Encompasses a broad range of checks designed to encourage adherence to established best practices in software development. This includes proper resource management, avoiding deprecated APIs, and ensuring code modularity.

    - **Documentation**: While not its primary focus, PMD can also help identify inadequacies in code documentation, such as missing or incomplete JavaDoc.

- **Ease of Use**
    - PMD is relatively easy to integrate and use in various development environments. It can be run as a standalone command-line tool, integrated into build tools like Maven and Gradle, or used within IDEs such as Eclipse, IntelliJ IDEA, and NetBeans through plugins.
    - Configuring PMD is straightforward, with support for custom rule sets that allow teams to focus on specific areas of interest or concern within their projects
    - The tool's output is clear, providing specific details about where issues are detected, the type of issue, and suggested fixes or considerations, making it actionable for developers