Structural-Based Testing Project

Praveen kumar Manoharan
July 2nd 2025

Table of Contents

Introduction

In this project we will be exploring the various tools to support structural testing for code coverage, decision coverage and Static code analysis.

Part1 - Statement & Decision Coverage Tool

For this project, I have selected "jacoco-maven-plugin", which is widely used plugin for code coverage and branch coverage. This tool is very easy to use to validate coverage with defining line coverage, branch coverage with the plugin's configuration of application pom.xml. This is very helpful in defining failing the build until the defined test coverage is implemented.

Plugin configuration

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.11</version>  <!-- Example version -->
  <executions>
    <execution>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>report</id>
      <phase>test</phase>
      <goals>
        <goal>report</goal>
      </goals>
       <configuration>
        <outputDirectory>${project.build.directory}/jacoco</outputDirectory>
      </configuration>
    </execution>
  </executions>
 </plugin>
```

Various configuration can be added with in the execution. In this above example I have created the reports with the output directory, which provides the html or xml-based output.

Reports generated by this tool are easy to understand the code coverage with different color coding in report. This helps developers to better understand line/branch which they were covered with their test. Also gives developer good understanding on what lines or branch missed in the test and cover them.

As stated above this tool provides a very easy configuration to control the project requirements on line and branch coverage in build, so build can be failed if the required coverage is not attained.

VendingMachine.java Test cases

Listed below the test case used in for the sample vendingMachine class

1. TestDispenseCandyExact - Test cases covered to validate the candy with equal input cost price of 20.
2. TestDispenseCokeExact – Test case to validate the coke with equal input cost price of the item.
3. TestDispspenseCoffeExact – Test case to validate item coffee with cost greater than coffee cost, which was validated for "Item dispensed and change".
4. TestNotEnoughForAny – Test case to cover all conditions and check for nothing can be purchase.

5. TestEnoughforcandyonly – Test case to validate the input provided as part of validation to get only Candy can be purchase.
6. TestEnoughforCandyandCoke – Test case to validate the input provided and update that the provide input only can purchase candy and coke.

Test case - code

```
public class VendingMachineTest {

    @Test
    public void testDispenseCandyExact() {
        String result = VendingMachine.dispenseItem(input:20, item:"candy");
        assertTrue(result.contains(s:"dispensed") || result.contains(s:"candy"));
    }

    @Test
    public void testDispenseCokeExact() {
        String result = VendingMachine.dispenseItem(input:25, item:"coke");
        assertTrue(result.contains(s:"Item dispensed") || result.contains(s:"coke"));
    }

    @Test
    public void testDispenseCoffeExact() {
        String result = VendingMachine.dispenseItem(input:50, item:"coffee");
        assertTrue(result.contains(s:"Item dispensed and change") || result.contains(s:"coffee"));
    }

    @Test
    public void testNotEnoughForAny() {
        String result = VendingMachine.dispenseItem(input:10, item:"candy");
        assertTrue(result.contains(s:"Cannot purchase item."));
    }

    @Test
    public void testEnoughForCandyOnly() {
        VendingMachine vm = new VendingMachine();
        String result = VendingMachine.dispenseItem(input:22, item:"coke");
        assertTrue(result.contains(s:"Can purchase candy."));
    }

    @Test
    public void testEnoughForCandyOrCoke() {
        String result = VendingMachine.dispenseItem(input:38, item:"coffee");
        assertTrue(result.contains(s:"Can purchase candy or coke."));
    }

}
```

Jacoco Generated Html report

project4 > default > 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

VendingMachine.java

```
1.  /* Java program for Vending Machine. The class takes in the 2 parameters
2.  and returns whether the item can be dispensed or not */
3.
4.  public class VendingMachine
5.  {
6.
7.      public static String dispenseItem(int input, String item)
8.      {
9.          int cost = 0;
10.         int change = 0;
11.         String returnValue = "";
12.     ◆   if (item == "candy")
13.             cost = 20;
14.     ◆   if (item == "coke")
15.             cost = 25;
16.     ◆   if (item == "coffee")
17.             cost = 45;
18.
19.     ◆   if (input > cost)
20.         {
21.             change = input - cost;
22.             returnValue = "Item dispensed and change of " + Integer.toString(change) + " returned";
23.         }
24.     ◆   else if (input == cost)
25.         {
26.             change = 0;
27.             returnValue = "Item dispensed.";
28.         }
29.         else
30.         {
31.             change = cost - input;
32.         ◇   if(input < 45)
33.                 returnValue = "Item not dispensed, missing " + Integer.toString(change) + " cents. Can purchase candy or coke.";
34.         ◆   if(input < 25)
35.                 returnValue = "Item not dispensed, missing " + Integer.toString(change) + " cents. Can purchase candy.";
36.         ◆   if(input < 20)
37.                 returnValue = "Item not dispensed, missing " + Integer.toString(change) + " cents. Cannot purchase item.";
38.         }
39.
40.         return returnValue;
41.
42.     }
43. }
```

With the above test we are able to cover 100% for statement coverage and 93% of branch coverage.

Also, these reports provide a clearer explanation of the missing branch coverage, when we hover on the yellow content in the html report.

From developer tools like eclipse, inteliJ, visual code - we can download the plugins for Jacoco to verify the coverage with running tests on the application or individual class. Also, this tool provides report merging, which can be aggregated from lower modules to higher project, which will be of super helpful for developers working multiple module applications.

Part2 – Static code Analysis tool

For static code Analysis tool, I have selected "**checkStyle**". As part of this check style, we can capture various data anomalies in code using static analysis like naming conventions, modifier, code indentations, cleaner import, avoid unused imports, code comments, StringliteralEquality etc. This will help developer to capture the anomalies in the early stage of the development with in local build and fix them.

For this example, for static code analysis, I have created the checkstyle.xml and configured them as part of the build to perform static analysis for my project. In this check style I have performed stringliteralEquality, unusedLocalvariable check, naming conversion check. This is to validate the code as any unused local variable, string literal equality to make sure we are using ***equals*** and avoiding "==" in string comparison. Also, this tool provides flexibility in configuring the needs of the project, for better application validation on various anomalies.

Feature and functionalities

- Coding standards Enforcement.
- Custom rules configuration through XML
- Integration available for Maven, gradle and developer tools (IDEs).
- Supports XML report & HTML reports

Coverage Provided

This tool provides the below coverage

- String comparison
- Unused local variables & imports
- Javadoc presences for classes and methods
- Naming conventions for classes, methods, variable and constants.
- Coverage details are easily generating XML and HTML reports for review.

Ease of use

- Easy to integrate with Maven or Gradle builds
- It's very easy to configure rules, for the project via the XML file loaded in the configLocation in pom.xml.
- This plugin gives us ability to fail the build or provide warning.
- Easy generation of XML or HTML reports for review.

Configuration of the plugin in maven Pom

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-checkstyle-plugin</artifactId>
    <version>3.6.0</version>
    <configuration>
     <configLocation>checkstyle.xml</configLocation>
     <failsOnError>false</failsOnError>
     <consoleOutput>true</consoleOutput>
     <violationSeverity>warning</violationSeverity>
     <outputFile>${project.build.directory}/checkstyle/checkstyle-report.xml</outputFile>
    </configuration>
    <executions>
     <execution>
      <id>checkstyle-validation</id>
```

```
        <phase>validate</phase>
        <goals>
          <goal>checkstyle</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
```

Anomalies used to validate Static Code Analysis. As part of the process, we validate the application code on static bases to make sure the system exceptions are catches at the earliest.

Checkstyle configuration

```
<?xml version="1.0"?>
<!DOCTYPE module PUBLIC "-//Checkstyle//DTD Checkstyle Configuration 1.3//EN"  "https://checkstyle.org/dtds/configuration_1_3.dtd">
<module name="Checker">
  <module name="TreeWalker">
    <!-- Enforce using .equals() for string comparison -->
    <module name="EqualsAvoidNull"/>
    <!-- Disallow unused local variables -->
    <module name="UnusedLocalVariable"/>
    <!-- Enforce variable naming conventions -->
    <module name="LocalVariableName"/>
    <!-- Enforce class naming conventions -->
    <module name="TypeName"/>
    <!-- Enforce method naming conventions -->
    <module name="MethodName"/>
    <!-- Enforce constant naming conventions -->
    <module name="ConstantName"/>
    <!-- Enforce indentation -->
    <!-- <module name="Indentation">
      <property name="basicOffset" value="4"/>
      <property name="braceAdjustment" value="0"/>
      <property name="caseIndent" value="4"/>
      <property name="tabWidth" value="4"/>
    </module> -->
    <!-- Enforce whitespace rules -->
    <!-- <module name="WhitespaceAfter"/>
    <module name="WhitespaceAround"/> -->
    <!-- Enforce Javadoc for classes and methods -->
    <module name="JavadocType"/>
    <module name="JavadocMethod"/>
    <module name="UnusedLocalVariable"/>
    <module name="EqualsAvoidNull"/>
    <module name="UnusedImports"/>
    <module name="StringLiteralEquality"/>
  </module>
</module>
```

Report

Checkstyle generates the html or XML report on the code analysis configured in the maven pom. In this report the rules executed for this project are listed and the errors which occurred are listed in the report.

- Unused Local variable weight &length
- String literals should be compared with equal and not ==

project4

Last Published: 2025-07-02 | Version: 1.0-SNAPSHOT project4

Checkstyle Results

The following document contains the results of Checkstyle 9.3 with checkstyle.xml ruleset.

Summary

Files	Info	Warnings	Errors
2	0	0	6

Files

File	I	W	E
StaticAnalysis.java	0	0	3
VendingMachine.java	0	0	3

Rules

Category	Rule	Violations	Severity
coding	StringLiteralEquality	4	Error
	UnusedLocalVariable	2	Error
	UnusedLocalVariable	2	Error

Details

StaticAnalysis.java

Severity	Category	Rule	Message	Line
Error	coding	UnusedLocalVariable	Unused local variable 'weight'.	16
Error	coding	UnusedLocalVariable	Unused local variable 'length'.	17
Error	coding	StringLiteralEquality	Literal Strings should be compared using equals(), not '=='.	25

VendingMachine.java

Severity	Category	Rule	Message	Line
Error	coding	StringLiteralEquality	Literal Strings should be compared using equals(), not '=='.	13
Error	coding	StringLiteralEquality	Literal Strings should be compared using equals(), not '=='.	15
Error	coding	StringLiteralEquality	Literal Strings should be compared using equals(), not '=='.	17

Built by: maven