Comparing Structural Coverage Tools

Mayisha Farzana
Department of Electrical and Computer Engineering
University of Ottawa
mayishafarzana3@cmail.carleton.ca

Abstract— ***This project aims to analyse each subject program with different structural coverage tools and compare them differently. To run the project, I selected Java to construct the project because it works across multiple platforms and has adequate test coverage tools. I executed each subject program with the test suite and measured the structural coverage. After gathering all the data using the tools, I obtained the various coverage reports for each subject programme and compared and explained their quantitative and qualitative differences. Moreover, I explained why the coverage reports differ for each subject program when changing coverage tools and the understanding of each coverage.***

Keywords—structural coverage tools, subject program, jacoco, open clover, IntelliJ IDEA.

Introduction

Coverage tools are really essential for measuring software quality and are an important component of software maintenance. It gives statistics on various coverage elements, which helps us to evaluate the testing effectiveness. It is mainly a measure, not a method or test. We mainly use the coverage in the testing part. It also improves testing by designing additional test cases for untested code, increasing code quality, increasing code coverage and decreasing costs. For example, if we get 80% code coverage means 20% of the code is not covered under the tests. It is a quantitative measure of test coverage, an indirect quality check method. It can also ensure the quality of the test.
I used Java as a programming language to perform the project because I got some open-source coverage tools here. Code coverage tools integrate with the build tools like Ant, Maven, Gradle, CI tools like Jenkins, and a variety of other tools that make up the software development toolset.
The goal of this project is to use different coverage tools and check why each coverage report's results differ from each other for each subject program.
Here, I will discuss five subject programs with the three coverage tools I have used for running my Java project, which was easy to use. I will also discuss one coverage tool I tried to use, but due to some error in the code, I could not get the coverage report.

I. Selection of Coverage Tools

Here, I will discuss the four coverage tools I have used for running my Java project.

A. Jacoco:

JaCoCo is a free code coverage toolkit developed by EclEmma. In Eclipse, we can get this EclEmma under the Eclipse Public License. To download this, we can download it from the Eclipse Marketplace. In Jacoco, only Java-based applications which require reporting and measurement can be used.

Choosing Jacoco as a coverage tool:

Jacoco is a newer coverage tool, and it is under the active development process for which its work is progressing day by day. It is a pure Java tool and does not require a specific configuration. We can download it from Eclipse Marketplace and use this coverage tool in the code. Moreover, it also supports a broader range of coverage metrics and is generally faster in terms of performance. It also uses an efficient algorithm for calculating coverage data. In terms of code coverage, it includes lines, instructions, methods, type, branching, and cyclomatic complexity. It is easy to configure JaCoCo in Maven, Junit etc., to get a code coverage report.

B. Open Clover:

Open Clover is also an open-source coverage tool. It's a commercial tool from Atlassian, a popular JIRA software company. It provides applications and IDE plugins for both IDEs, including IntelliJ IDEA and Eclipse.

Choosing Open Clover as a Coverage Tool:

Open Clover coverage can run on Java applications. Each project's open clover coverage metrics tool can be the method, statement, branch, global coverage, and per-test coverage. It also has built tool integrations such as command line, Ant, Maven, Grails, and Gradle. It also quickly measures the testing coverage. It also supports various formats for generating a report, including text, HTML, XML, PDF, and JSON.

C. IntelliJ IDEA:

Intellij IDEA is a code coverage tool built-in code coverage tool in the IntelliJ IDEA. We can see how much of our code has been executed from it. It also enables us to determine the degree to which our code is covered by unit tests, allowing us to gauge the efficacy of these tests.

Choosing IntelliJ IDEA as a coverage tool:

It supports Maven, Java, Gradle and Git. One of the great features is that we can run code coverage when developing the code before pushing it to a remote SCM to run any code

analysis tool like SonarCube. In Eclipse, we may have to use some plugin or give the path of the coverage tool to use it on the project, but in IntelliJ, it is already built. After downloading the project, we can start running it and see the coverage report in various formats like pdf and HTML.

D. Cobertura:

It is a reporting tool which mainly calculates the test coverage for the programs for each java project; they calculate the percentage of branches/lines which are accessed by unit tests.

Choosing Open Clover as a Coverage Tool:

It can use to pinpoint the areas of your Java programme where tests are not sufficiently covered. Its foundation is jcoverage. Without the source code, it can measure coverage and is simple to use. It can test classes, method lines, and branches and displays report in HTML or XML format.

Comparison of Code Coverage Tools:

	Jacoco	Open Clover	IntelliJ IDEA	Cobertura
	Build Tools Integration			
Ant	✓	✓	✓	✓
Maven	✓	✓	✓	✓
	Coverage Metrics			
Method	✓	✓	✓	×
Statement	×	✓	✓	×
Line	✓	×	✓	✓
Branch	✓	✓	✓	✓
Global Coverage	✓	✓	✓	✓
	Report Type			
HTML report	✓	✓	✓	✓
XML	✓	✓	✓	✓
Supported Language: Java	✓	✓	✓	✓
	IDE Integrations			
IntelliJ IDEA	✓	✓	✓	×
Eclipse	✓	✓	✓	✓ eCobertura

Fig 2.1: Comparison of Code Coverage Tools

When we have the code coverage report, we can estimate which parts still need to be in unit-test and from seeing it, we can try to get full code coverage after changing or adding some test cases.

II. PROPOSED METHODOLOGY

In this section, I am going to describe the proposed methodology briefly. I will also demonstrate the working procedure using the coverage tools in the subject programs.

1) Coverage: Jacoco:

To run the subject program, I did not use the latest Eclipse version to run the project since most of the projects supported the older version. Here, from the marketplace, I downloaded the EclEmma and also added the JUnit to the project. For the JUnit, I also had to check which one supported the project. Before running the project, I had to change the Java version's environment variable. Later, I ran the project with coverage with test cases and got the generated HTML report after finishing the overall project.

2) Coverage: Open Clover:

To run the project, I used Eclipse 1.15.0. It is the same version and platform I used to find the coverage for Jacoco. Here, I had to download the open clover from the install new software. Then, I had to enable open clover for this project and add the JUnit from the add library. After that, I had to run the whole program as Clover coverage.

3) Coverage: IntelliJ IDEA:

Most of my subject programs were Maven-supported. To run the project, I downloaded the latest version of IntelliJ IDEA Community 2022.2.2 to run this project and used Java version 11 for it. After downloading the Maven, I had to include it in the project. In IntelliJ, we always mention which folder contains the source files and which contains the test files. Here, in the project, we need to add the necessary Java library to run the project. After that, run all the test cases with coverage and got the report in HTML format.

4) Coverage: Cobertura:

Cobertura is a coverage tool for Java. To download the Cobertura, I downloaded it from the link.

Link: https://github.com/cobertura/cobertura

Since Cobertura can only be used in the command line or via ant tasks, and most of my projects supported Maven, I faced some errors here. To run my projects since I could not use Ant, I wrote a command line to run the project. In the instruction part, it was mentioned to do the instrumenting part.

Instructions Given:

cobertura-instrument.bat

[--basedir dir] [--datafile file] [--auxClasspath classPath] [--destination dir] [--ignore regex] classes [...]

What I wrote:

cobertura-instrument.bat --destination D:\Fall 2022\build\instrumented D:\Fall 2022\build\classes

Running Tests:

I initially needed to download cobertura.jar files to run the test cases. Later I had to add the directory which contains the instrumented class. It was mentioned that before the directory, which contains the uninstrumented classes, I needed to

instrumented the classes which contain the directory. The orders are important here. I did not need to include JUnit's here. When instrumented classes can access, the Cobertura updates their data file. After doing all these things, I wrote the command lines in the command prompt to run the project. Java -cp C:\cobertura\lib\cobertura.jar;D:\Fall2022\build\instrumented; D:\Fall2022\build\classes; D:\Fall2022\build\test-classes -Dnet.sourceforge.cobertura.datafile=D:\Fall2022\build\cobertura.ser ASimpleTestCase (Class Name)
I included the test files, and later I got an error while running it.

III. DESCRIPTION OF SUBJECT PROGRAMS

Now, I will describe the five subject programs, how I implemented them, what challenges I faced and how I solved them.

A. Project Name: Jtopas

The Jtopas project mainly offers a compact, user-friendly Java solution for the frequent issue of parsing random text input. The source for this data is various programming languages, a stream of HTML, XML, or RTF, or a basic configuration file with a few comments. It is an open-source project.
In this subject program, we have used three different coverage tools.

	Jacoco	Open Clover	IntelliJ IDEA
Platform	Eclipse 1.15.0	Eclipse 1.15.0	Intelli 2020
Java Version	11	15	11

1) Coverage: Jacoco:
I used Ant to run this project in Eclipse 1.15.0. because Ant is the build of Java applications which supplies several built-in tasks allowing to compile, assemble, test and run Java applications.
Problems Faced: I did not use the latest Eclipse version to run this project since it was an old version and was modified around 2004. After that, the source project did not upgrade their project, and in the latest version of Eclipse, the version needed to be compatible. That's why I use Eclipse 1.15.0 version.
To match the version, I chose Java Version 11 and JDK 15, which worked to run the code and got the coverage report. I also tried to run it in Java Version 15, and since it was a higher version, it was showing an error. Later, I selected a lower version of Java 1.5.0._22, and it was a lower version and was not working here. So, Java Version 11 matched with the projects to run in the environment. For this, I had to change the environment variable.
After adding the JUnit 5, I run the code to get the coverage report for Jacoco.
Result: Here, 85% of the instruction coverage has been executed in the overall code. I can understand which logical paths got tested or not from branch coverage, contributing to a more comprehensive test suite. Here, it covered 58% of the branch coverage.

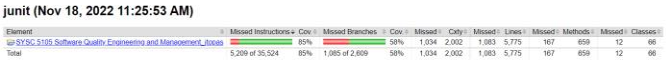


Fig 3.1: The coverage report for Jtopas using Jacoco Coverage Tool

2) Coverage: Open Clover:
Problems Faced: Initially, I chose JUnit 5, but it showed an error. Then I selected JUnit 3 in the clover configuration and set JUnit 3 to test all the test cases in the project, and then I got the coverage report in the HTML format.
Results: Here, I got 76% of total code coverage, which does not mean that 76% of the lines are covered; it means 76% of the code tested throughout the project

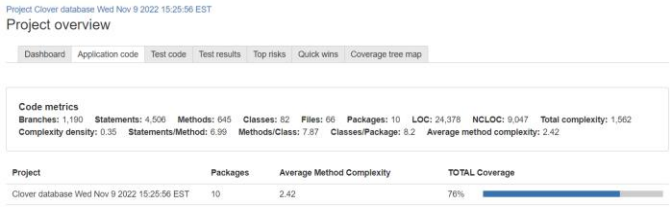


Fig 3.2: The coverage report for Jtopas using Open Clover Coverage Tool

3) *Coverage: IntelliJ IDEA:*
Here, it runs in Maven, and this tool helps get the right JAR files for each project, as there may be different versions of separate packages. I have downloaded the latest version of IntelliJ IDEA community 2022.2.2 to run this project and used Java version 11 for it.
Here, for this project, the JUnit folder contains the test folder, and the Src folder contains the sources folder. After setting these two, to run the project, I had to download JUnit 4, JUnit 3 and hamcrest and add it to the building structure.
Problems Faced: Here, in the building structure, I had to add JUnit 4 and hamcrest at first. It is the Java library because without adding it, I ran it, and I saw in some JUnit frameworks it was showing an error. So, after adding it, it resolved, but later, I got another error for another java framework. Then I downloaded JUnit 3, and the overall project ran this time, and I got the coverage report for this project.
Results: For all classes, it covered 78% of the coverage report, 75.2% of the method and 72.5% of the lines covered.

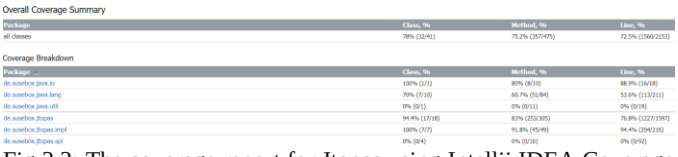


Fig 3.3: The coverage report for Jtopas using Intellij IDEA Coverage Tool

B. Project name: XML-Security:

XML security means to standard security requirements of XML documents such as confidentiality, integrity, message authentication, and non-repudiation. It is an open-source project.

	Jacoco	Open Clover	IntelliJ IDEA

Platform	Eclipse 1.15.0	Eclipse 1.15.0	IntelliJ 2020
Java Version	11	15	11

In this subject program, we have used three different coverage tools.

1) Coverage: Jacoco:
For this project, I used Maven because it helps to download dependencies, which refer to libraries or JAR files.
While running the project, I was getting the error 'No tests found with test runner JUnit 5'. This is why I needed help getting the coverage report. Later, I added JUnit 5 in the build path and some external libraries; so I could run the project.
Result: Instruction Coverage 79%, Branches Coverage: 61%.

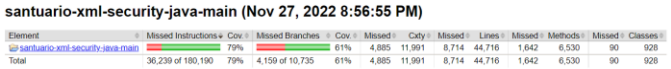


Fig 3.4: The coverage report for XML-Security using Jacoco Coverage Tool

2) Coverage: Open Clover:
To run the project, I added JUnit 5 to run the project. In every project, I included JUnit to create an up-to-date foundation for developer-side testing on the JVM. This includes focusing on Java 8 and above, as well as enabling many different styles of testing.
Here, I got 66.9% of the total coverage, meaning the application code is exercised when an application is running.

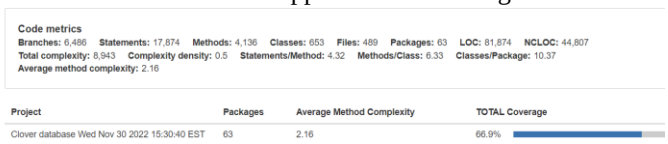


Fig 3.5: The coverage report for XML-Security using Open Clover Coverage Tool

3) Coverage: IntelliJ IDEA:
To run the project in IntelliJ IDEA, I had to mention which one is the source folder and which one is the test source's root. Here, in the source folder, I got one folder named main, my source folder, and another folder named test, which was my test folder. After mentioning that, then I run with all test coverage.
Initially, I just remembered that I needed to clean the project and faced some trouble. Later, when I reloaded it again or wrote a command, I could run the project and get a coverage report. Here, it covered 93.2% of the class, 74.6% of the method and 72% of the line.

Fig 3.6: The coverage report for XML-Security using Intellij IDEA Coverage Tool

C. *Project name: Janino:*

Janino project is a Java compiler. It generates JavaTM bytecode that is loaded and executed directly. It is also an open-source project.

	Jacoco	Open Clover	IntelliJ IDEA
Platform	Eclipse 1.15.0	Eclipse 1.15.0	IntelliJ 2017.3.7
Java Version	11	1.7	1.7

In this subject program, we have used three different coverage tools.

1) Coverage: Jacoco:
To run the project, I used Eclipse 1.15.0 version, checked the Java Version and added JUnit in the folder from the build path. Later I ran the project and got the coverage report. I got 54% for instruction coverage and 44% for branch coverage.

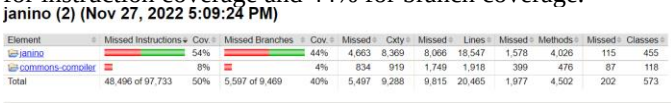


Fig 3.7: The coverage report for Janino using Jacoco Coverage Tool

2) Coverage: Open Clover:
In Janino, initially, I needed help getting the coverage report for open clover. It was showing some error. Later, I discovered it was showing an error for the Java version. I used Java 1.7 version for it, and it supported this project.
In open clover, I got the low coverage which is 53.4%.

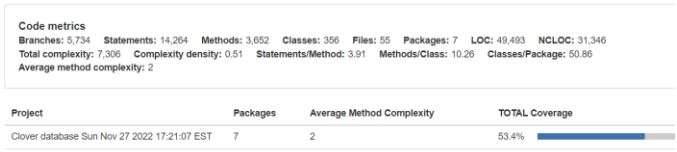


Fig 3.8: The coverage report for Janino using Open Clover Coverage Tool

3) Coverage: IntelliJ IDEA:
To run this Janino project in Intellij, I had to use an older version, 2017.3.7, because it was showing an error in the latest version. Since this Janino project is old, it is compatible with the older version. I also used Java 1.8 version to run the project. It does not support Java's latest version. Previously, I tried with Java 11 and Java 15 versions, but some files were showing errors. After using Java version 1.8, I got the result. It is also a Maven project. In the coverage report, I got 64.5% for class coverage, 55.2% for method coverage and 52.2% for line coverage.

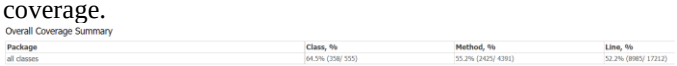


Fig 3.9: The coverage report for Janino using Intellij IDEA Coverage Tool

D. *Project name: Jacoco:*

JaCoCo stands for Java Code Coverage. The EclEmma team created this free code coverage library. It creates code coverage reports and integrates well with IDEs like IntelliJ IDEA, Eclipse IDE, etc. It is an open-source toolkit, and I have also used this as coverage for my subject programs. Since this project file is small, it is easy to run this file faster.

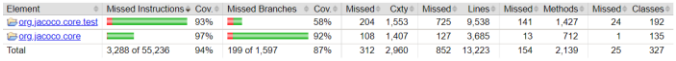
	Jacoco	Open Clover	IntelliJ IDEA

Platform	Eclipse 1.15.0	Eclipse 1.15.0	IntelliJ 2022.2.2
Java Version	15	15	15

1) Coverage: Jacoco:

Jacoco Maven plug-in provides the JaCoCo runtime agent to our tests and allows basic report creation. Initially, I used Eclipse 1.15.0 version to run the Jacoco project. Since this project builds to run the Jacoco code coverage tests, I got good coverage results for this project.

I got 97% for the instruction coverage and 92% for the branch coverage.



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.jacoco.core.test		93%		58%	204	1,553	725	9,538	141	1,427	24	192
org.jacoco.core		97%		92%	108	1,407	127	3,685	13	712	1	135
Total	3,288 of 55,236	94%	199 of 1,597	87%	312	2,960	852	13,223	154	2,139	25	327

Fig 3.10: The coverage report for Jacoco using Jacoco Coverage Tool

2) Coverage: Open Clover:

To run the project, we need to enable the clover option in the project.

Before running the whole project here, we need to specify jacoco.core.test where all the test cases are. After selecting this test folder in configurations and then setting Junit, we need to run the coverage. Jacoco only runs in core.test and core file. The whole project will not run for the test cases here.

From here, I got 95.8% of the total coverage for the project.

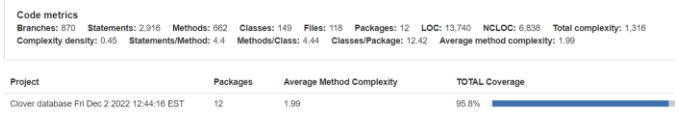


Fig 3.11: The coverage report for Jacoco using Open Clover Coverage Tool

3) Coverage: IntelliJ IDEA:

To run the Jacoco file, I used IntelliJ 2022 latest version and Java version 15. Jacoco.core.test is our test source's root, and Jacoco.core is the root of the source for this project. After mentioning it, I will run the project and

I got 40.9% class coverage, 46.6% method coverage and 54% line coverage.

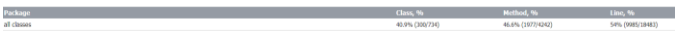


Fig 3.12: The coverage report for Jacoco using Intellij IDEA Coverage Tool

E. Project name: Guava:

Many of the Google core libraries we use in our Java-based projects, such as collections, caching, primitives support, concurrency libraries, common annotations, string processing, I/O, and others, are contained in the Guava project. It is also an open-source project, and it is a large project where it contains 88559 LOC.

	Jacoco	Open Clover	IntelliJ IDEA
Platform	Eclipse 1.15.0	Eclipse 1.15.0	IntelliJ 2022.2.2
Java Version	15	15	15

1) Coverage: Jacoco:

To run the project, I added JUnit 5 here and then ran the project with the jacoco code coverage. In the run configuration, set guava-tests/test as a test folder to run the test coverage. After running it, I got 80% of instruction coverage and 82% of branch coverage.

When I was running the code after setting JUnit 5, I was getting an error. Then in the run configuration, I saw that I needed to set the test cases and which folder contains them, then needed to run it. After doing it, I could run the project and get the code coverage report.

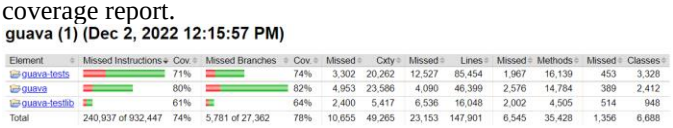


Fig 3.13: The coverage report for Guava using Jacoco Coverage Tool

2) Coverage: Open Clover:

To run the project as open clover, I need to enable clover in the project. After setting the JUnit and the test cases, I need to run the files as clover and will get the coverage report. I got 89.8% as total code coverage.

Fig 3.14: The coverage report for Guava using Open Clover Coverage Tool

3) Coverage: IntelliJ IDEA:

To run in Intellij, I need to set the sources root and test sources root. In the Guava folder, tests are set as the test folder, and the main folder is set as the source root folder. Then, after setting it, I ran the project with the test coverage and got the result. It covers 32.8% class, 12.9% method and 10.6% line. Since the project was big, the percentage of coverage could be better. It also takes a huge amount of memory to run the project. Initially, after passing some of the test cases, it crashed and got hung. Later, after cleaning and restarting it, I got the coverage report result.

Fig 3.14: The coverage report for Guava using Intellij IDEA Coverage Tool

IV. RESULTS ANALYSIS

I compared the subject programs with the structural coverage tools and showed their results. I also mentioned why their results vary.

Comparing All The Coverage Tools Using The Subject Programs:

Coverage Tools	Jacoco	IntelliJ IDEA	Open Clover
Subject Program	**JTOPAS**		
Methods	74.65% (492/659)	75.2% (357/475)	645
Lines	81.24% (4692/5775)	72.5% (1560/2153)	24378
Classes	81.81% (54/66)	78% (32/41)	82
Subject Program	**XML-Security**		
Methods	75.84% (4888/6530)	74.65% (2762/3703)	4,136
Lines	80.51% (36002/44716)	72% (12918/17941)	81,874
Classes	90.30% (838/ 928)	93.2% (465/499)	653
Subject Program	**Janino**		
Methods	60.80% (2448/4026)	55.2% (2425/4391)	3652
Lines	56.51% (10,481/18,547)	52.2% (8985/17212)	49,493
Classes	74.73% (340/455)	64.5% (358/555)	356
Subject Program	**Jacoco**		
Methods	98.17% (699/712)	46.6% (1977/4242)	662
Lines	96.55% (3558/3685)	54% (9985/18483)	13,740
Classes	99.26% (134/135)	40.9% (300/734)	149
Subject Program	**Guava**		
Methods	82.58% (12208/14784)	12.9% (2139/16544)	5448
Lines	91.19% (42309/46399)	10.6% (5130/48392)	88559
Classes	83.87% (2023/2412)	32.8% (780/2381)	740

Fig 4.1: Comparing all the coverage tools using the subject programs

Comparing the three coverage tools, we can discuss some findings from there. For the jtopas project, we can see that IntelliJ covers 75.2%, whereas Jacoco covers 74.65%, which is less than that. For classes, jacoco covers 81.81%, which means it covers 54 out of 66 classes. On the other hand, IntelliJ, covers 78%, which is 3.81% less than that. Open clover runs 82 classes, but we do not know how much it covers fully. Here, we can see Jacoco performs better than Intellij IDEA.

However, for XML-Security, using jacoco, it covers 36002 lines out of 44716 lines which means 80.51%, whereas IntelliJ covers 72%, which is around 8.51% lower than that. Here, we can also see Jacoco performs better than Intellij IDEA.

In Intellij, for methods in the janino project, it covers around 55.2%, whereas, for jacoco, it covers 60.80%, which is 5.6% higher than that. For Open clover, it covers 3652 methods. As a result, Jacoco performs well than other coverage tools.

We get 99.26% class coverage for Jacoco project when using jacoco coverage tools whereas we get only 40.9% class coverage when using IntelliJ IDEA. Here, we can see a big difference in the coverage results and here, we also get good coverage for jacoco.

Lastly, Using Guava, for open clover covers only 740 classes, whereas jacoco covers 2023 classes out of 2412, which performs better.

From here, we can conclude that comparing these three coverage tools, Jacoco performs better than the two others.

Comparing The Subject Programs with The Coverage Tools:

SUBJECT PROGRAM NAMES	COVERAGE TOOL: JACOCO	
	Instructions	Branches
Jtopas	85%	58%
Xml-Security	79%	61%
Janino	54%	44%
Jacoco	**97%**	**92%**
Guava	80%	82%

Fig 4.1: Using Jacoco Coverage report comparing the subject programs

The above table shows that we got the highest percentage of coverage in the Jacoco project when using the Jacoco coverage tool. It covered 97% of the instructions and 92% of the branches. Because Jacoco tested their project with the jacoco coverage tool, it got the highest coverage tool since it was built and designed according to the jacoco project. On the other hand, Janino got the lowest percentage of coverage when using Jacoco coverage tool. It got 54% of the instructions and 44% for the branches. In most of the subject programs, we got a better percentage in instructions than in the branches. This can be explained in an example that will be clearer to us.

```
if (condition) {
   if (condition2) {
      System.out.println("Hey!");
   }
}
```

In this test case, both condition and condition2 are true, and the string will be printed "Hey!" on the screen. For it, all the statements will be executed, and the coverage will be 100%. But on the other hand, if condition2 is false which means the string will not be displayed in the screen. Since the condition is false, this will not be exercised through the test. From it, branch coverage carries more information, which is why here, in the coverage report, we got a lower percentage than the instruction coverage.

SUBJECT PROGRAM NAMES	COVERAGE TOOL: JACOCO			
	Missed	Cxty	Missed	Lines
Jtopas	1034	2002	1083	5775

Xml-Security	4885	11991	8714	44716
Janino	4663	8369	8066	18547
Jacoco	108	1407	127	3685
Guava	4953	23586	4090	46399
	Missed	Methods	Missed	Classes
Jtopas	167	659	12	66
Xml-Security	1642	6530	90	928
Janino	1578	4026	115	455
Jacoco	13	712	1	135
Guava	2576	14784	389	2412

Fig 4.2: Using Jacoco Coverage report comparing the subject programs

In JaCoCo, it calculates cyclomatic complexity (Cxty) for each non-abstract method and summarizes complexity for classes, packages and groups. Here, jacoco calculates cyclomatic complexity using one equation based on the number of branches (B) and the number of decision points (D):

$v(G) = B - D + 1$

Based on each branch's coverage status, JaCoCo also calculates covered and missed complexity for each method. Missed complexity again indicates the number of test cases missing to cover a module fully.

Here, in the subject program of xml-security, we can see that the cxty is 11991, where it missed 4885. On the other hand, in Jacoco, out of 1407 cxty, it is missed only 108.

In the Guava project, they all had 46399 lines where they missed coverage of 4090 lines. For the jtopas, among 659 altogether methods, they missed covering 167 methods in the test cases. Moreover, in jacoco, they did not miss many classes among 135 classes, they only missed one class to cover while doing the test cases, and it is the least missed coverage for lines. On the other hand, in Janino project, out of 4026 methods, they failed to cover 1578 test cases while doing coverage tests. Since Jacoco has the highest coverage, it missed the project's minor lines, methods and classes.

SUBJECT PROGRAM NAMES	**COVERAGE TOOL: OPEN CLOVER**
	Total Coverage
Jtopas	76%
Xml-Security	66.9%
Janino	53.4%
Jacoco	**95.8%**
Guava	89.8%

Fig 4.3: Using Open Clover Coverage report comparing the subject programs

Here, in open clover, Jacoco gets the highest percentage of code coverage, 95.8%, whereas we get the lowest percentage of coverage in xml-security and which is 66.9%.

Coverage Tool: Open Clover	Jtopas	Xml-security	Janino	Jacoco	Guava
Branches	1190	6486	5734	870	5766
Statements	4506	17874	14264	2916	16243
Methods	645	4136	3652	662	5448
Classes	82	653	356	149	740
Files	66	489	55	118	377
Packages	10	63	7	12	18
LOC	24378	81874	49493	13740	88559
NCLOC	9047	44807	31346	6838	44904
Total Complexity	1562	8943	7306	1316	9092
Complexity density	0.35	0.5	0.51	0.45	0.56
Statements	6.99	4.32	3.91	4.44	7.36
Classes	7.87	6.33	10.26	12.42	41.11
Avg. Method Complexity	2.42	21.16	50.86	1.99	1.67

Fig 4.4: Using Open Clover Coverage report comparing the subject programs

These code metrics show how many of the total branches, statements, and methods are there in the project. Here, the Guava project has a total of 88559 lines of code in the project and was in the cove coverage; it covered around 89.8%. From the code coverage, we can identify which areas of the source code remained untested/uncovered by the tests.

SUBJECT PROGRAM NAMES	**COVERAGE TOOL: INTELLIJ IDEA**		
	Class	Method	Line
Jtopas	78%	75.2%	72.5%
Xml-Security	**93.2%**	**74.6%**	**72%**
Janino	64.5%	55.2%	52.2%
Jacoco	40.9%	46.6%	54%
Guava	32.8%	12.9%	10.6%

Fig 3.5: Using IntelliJ IDEA Coverage report comparing the subject programs

Here, within the IntelliJ IDEA code coverage, the xml-security subject program got the highest percentages of coverage than other subject programs. For class, it covered 93.2%; for method, it covered 74.6%; for lines, it covered 72%. For the Guava project, we got the lowest percentage of coverage. This is because Guava was a large project among these projects, and they could not cover many of the test cases in the whole program. In every subject program, we can see that most classes covered the highest percentage of coverage than method and lines covered. Class coverage mainly includes all classes in the package of classes that I am testing. We can see it from an example.

I have two classes in the package xml
-xml
  -ClassA.java
  -ClassB.java
for test
-xml
  -ClassATest.java
If I run test cases of ClassATest.java then I will get Class coverage 50% (1/2). Since both classes are in the same package, it included ClassB too.

But, if I create a ClassB object in any test case and run the same ClassATest cases again, then I will get Class coverage 100% (2/2) since I used both classes. So, it is clear to us that the class covers much coverage.

On the other hand, Line coverage covers the actual lines of code. Here, in IntelliJ, it marks all the lines Green & Red for Covered & Uncovered, respectively, so from it, I can easily check which lines it has considered.

I can also see from an example the difference between method coverage and line coverage and why the method coverage percentages are more than the line coverage percentage.

```
if(A>=3000)          condition false          lines executed
        if(B<=200)
                return true;
        else(C>30)
                return true;
else{                condition true           lines executed
        if(D>=50)                             lines executed
                return true;                  lines executed
        }                                     lines executed
```

Here, we can see that all the methods executed to check the condition, whereas when the condition becomes false, they do not execute inside at all, which means they do not execute the lines while tested. Here, five lines were executed out of nine lines, which covered the line coverage. On the other hand, it checks all the test coverage methods.

V. CONCLUSION

From doing this project, I identified the code coverage for each subject program, and it helped me to understand if the entire project is well developed and maintained properly and also helped me to know how much of the source code is tested. From it, I can assess the quality of my test suite. In Test Driven Development, where the developer develops their tests before writing the code, their code coverage is very crucial. It is also designed to guide the Agile development process and assist the programmers in producing better code with fewer unnecessary lines. Here, in these circumstances, code coverage aids developers in creating better tests and keeps their code on track by highlighting code that deviates from the intended development scope. Here, I compared each subject program with the coverage tools and also compared why the differences were created. After comparing all the coverage tools, I got a better performance for Jacoco coverage tools than the other tools. Jacoco is also a good coverage tool since it supports Java 7 and Java 8. It was important to consider all available metrics when finding if the code was well covered. From doing, code coverage, we can see which parts are missing in the code coverage, and it helps the developers to write better test cases by seeing its report.

REFERENCES

[1] A. Mackay, "What is meant by Structural (Code) Coverage? | QA Systems," *www.qa-systems.com*. https://www.qa-systems.com/blog/what-is-meant-by-structural-code coverage/#:~:text=Structural%20code%20coverage%20is%20a (accessed Dec. 04, 2022).

[2] "testing - What is code coverage and how do YOU measure it?," *StackOverflow*.https://stackoverflow.com/questions/195008/what-is-code-coverage-and-how-do-you-measure-it

[3] "Test Coverage in Software Testing," *www.guru99.com*. https://www.guru99.com/test-coverage-in-software-testing.html

[4] Admin, "Structural Coverage Analysis (SCA)» TheCloudStrap," Apr. 05, 2021. https://thecloudstrap.com/structural-coverage-analysis-sca/ (accessed Dec. 04, 2022).

[5] "Top 15 Code Coverage Tools (For Java, JavaScript, C++, C#, PHP)," *Software Testing Help*. https://www.softwaretestinghelp.com/code-coverage-tools/ (accessed Dec. 04, 2022).

[6] "EclEmma - JaCoCo Java Code Coverage Library," *Jacoco.org*, Oct. 12, 2019. https://www.jacoco.org/jacoco/

[7] "OpenClover 4.2 : Clover Documentation Home," *openclover.org*.http://openclover.org/doc/manual/4.2.0/index--clover-documentation-home.html (accessed Dec. 04, 2022).

[8] "Code coverage using Intellij IDEA," *www.linkedin.com*. https://www.linkedin.com/pulse/code-coverage-using-intellij-idea-vivek-malhotra/ (accessed Dec. 04, 2022).

[9] A. Guéret, "The real benefit of 100% code coverage," *Technology @ OpenClassrooms*, Nov. 24, 2021. https://medium.com/openclassrooms-product-design-and-engineering/the-real-benefit-of-100-code-coverage-b2f30393d659 (accessed Dec. 04, 2022).

[10] "Code Coverage Types: Which Is the Best?," *LinearB*, Jan. 14, 2022. https://linearb.io/blog/code-coverage-types/

[11] "What is the difference between class coverage and line coverage while running unit tests in Android Studio?," *Stack Overflow*.

[12] "What does Cxty in Jacoco report mean? - programador clic," *programmerclick.com*. https://programmerclick.com/article/7238828355/ (accessed Dec. 04, 2022).