



Software Measurement (SOEN 6611) – Team C

Dear Professor,

Our team would like to submit the report based on the following content selected project links and descriptions, metric descriptions, steps for collecting the data, steps for analyzing the data, describing the results (e.g., descriptive summaries for collected metrics, results of correlation analysis etc.), related work.

In this paper, we did correlation analysis among the following six metrics: branch coverage, statement coverage, mutation score, McCabe complexity, code churns and code smells.

The following table is the information about our team members:

Name	Student ID	E-mail
Shivam Nautiyal	40090841	shivam.nautiyal20@gmail.com
Nikunj Arora	40104832	nikunjarora333@gmail.com
Karan Sharma	40080005	95sharma.karan@gmail.com
Sardar Mutesham Ali	40094168	sardarms9@gmail.com
Saikiran Alagatham	40103833	saikiran.ask007@gmail.com

Following is a link to the replication package in GitHub:

https://github.com/Shivamnautiyal20/SOEN6611_Project_Group_C

Thank you,

Team C

Analyzing the Correlation between different Software Metrics Using Open Source Systems

Karan Sharma
Concordia University
Gina Cody School of
Engineering and
Computer Science
95sharma.karan
@gmail.com

Sardar Mutesham Ali
Concordia University
Gina Cody School of
Engineering and
Computer Science
sardarms9@gmail.com

Nikunj Arora
Concordia University
Gina Cody School of
Engineering and
Computer Science
nikunjarora333@gmail.com

Sai Kiran
Concordia University
Gina Cody School of
Engineering and
Computer Science
saikiran.ask007@gmail.com

Shivam Nautiyal
Concordia University
Gina Cody School of
Engineering and
Computer Science
shivam.nautiyal20@gmail.com

Abstract- This report aims to analyze the correlation between different software metrics using various open-source systems. Correlation demonstrates the extent to which two metrics are related to each other. We start by defining the metrics which we will be analyzing. The open-source systems are very carefully selected based on their usage history, their applications and the vast data present in them. We select 4 such projects averaging over 100K LOC and perform our evaluations on them and their versions.

Each system is then processed using various tools, plug-ins such as Jacoco, CLOC and JDeodrant to collect data. These tools help in collecting data for metrics such as Statement Coverage, Branch Coverage, Mutation score, McCabe Complexity, Code Churns and Code Smells.

After the experiment and data analysis, we found out that if a program is having high complexity then it will have less branch as well as statement coverage because of the high number of executable paths. Similarly, branch coverage and statement coverage has a strong positive correlation with the mutation score of the test suites of the project. Also, our data shows that the branch and statement coverage of software has a negative correlation with the Code Smells of software. Furthermore, we found out that the Code Churn and Code Smell are negatively correlated with each other.

Keywords— *Software Metrics, Correlation Analysis, Software Measurement*

Abbreviations - *LOC Line of Code, JaCoCo Java Code Coverage*

I. INTRODUCTION

With all the advancements in science and technology, the software is becoming a very integral part of this progress. As the need to have sophisticated systems increases, there is a lot of development work done. This rapid increase in software sizes, structures and functionality causes a lot of issues related to maintainability. Having these software metrics can help in analyzing how maintainable, how efficient a particular software is[1]. Such insights help in decision making regarding the future enhancements of the software.

In this paper, we tend to analyze the correlation between different metrics. According to the development experience, we selected four open-source projects and six software measurement metrics. The description of the selected projects and metrics is discussed in the following sections. And the steps to collect data and analyze them have been mentioned in the subsequent sections. Finally, the correlation between the metrics and the

conclusion of this experiment is mentioned in the final section of this paper.

II. PROJECTS DESCRIPTION

We selected four open source java projects. Each of these is well documented, maintained, usage history and a wide range of tests. For analyzing the correlation, we only consider the java code from these projects. Two of these projects are well over 100K lines of code. We considered five different versions to collect data across them to find the correlation between the code smells and code churn at the version level.

Considering different versions helps in getting insights regarding the evolution process and to understand the motive behind the development activities.

Project 1: Apache Commons Math (186K LOC)

Commons Math Provides a productive environment for aggregation, testing and support of efficient Java implementations of commonly used mathematical and statistical algorithms. It is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language or Commons Lang [2].

The versions selected for our study are 3.2, 3.3, 3.4, 3.5 and 3.6.

Project 2: Apache Commons Configuration (67K LOC)

The Commons Configuration software library provides a generic configuration interface which enables a Java application to read configuration data from a variety of sources. Commons Configuration provides typed access to single, and multi-valued configuration parameters.

The project is built by maven and consists of many documents to track the problems and help solve them during analysis[3]. We have selected versions 2, 2.4, 2.5, 2.6 and 2.7 to conduct our metric evaluation.

Project 3: Apache Commons Digester (27K LOC)

Many projects read XML configuration files to provide initialization of various Java objects within the system. There are several ways of doing this, and the Digester component was designed to provide a common implementation that can be used

in many different projects. Basically, the Digester package lets you configure an XML -> Java object mapping module, which triggers certain actions called rules whenever a particular pattern of nested XML elements is recognized. A rich set of predefined rules is available for your use, or you can also create your own[4].

Usually, it is considered as a layer above the SAX xml parser API. This makes it easier to process the inputs. In this the evaluation we considered Versions 1.8, 2.0, 2.1, 3.0 and the latest of all 3.2.

Project 4: Jfree Chart (317K LOC)

JFree Chart is a comprehensive free chart library for the Java(TM) platform that can be used on the client-side (JavaFX and Swing) or the server-side (with export to multiple formats including SVG, PNG and PDF). [5]

JFree Chart contains 120k LOC of java code with continuous issue tracking on GitHub and sourceforge.net as well. We have selected mainly five versions of code i.e. 1.0.14, 1.0.15, 1.0.16, 1.0.17, 1.0.19 that have some of the variations between code.

III. METRICS DESCRIPTION

In order to measure selected projects, six different metrics have been selected to get a different perspective of the measurement. These six metrics have been selected are below :

Metric 1: Statement Coverage

Statement coverage is a white box testing technique, which involves the execution of all the statements at least once in the source code. It is a metric, which is used to calculate and measure the number of statements in the source code which have been executed. It can also be used to check the quality of the code and the flow of different paths in the program.

It can also be used to check the quality of the code and the flow of different paths in the program. Statement coverage count is how many statements are executed at least once during the test and thereby the more coverage percent it shows, the more opportunity to find the existing bug. In white-box testing, the concentration of the tester is on the working of the internal source code and flowchart or flow graph of the code [6].

Generally, source code has a wide variety of elements like operators, methods, arrays, looping, control statements, exception handlers. Based on the input values given to the program, some code statements are executed and some may not be executed. The goal of the statement coverage technique is to cover all the possible executing statements and path lines in the code.

Formula

Statement coverage=(No of statements Executed/Total no of statements in the source code) * 100

Tool: JaCoCo

Metric 2: Branch Coverage

Branch coverage is a testing method, which aims to

ensure that each one of the possible branches from each decision point is executed at least once and thereby ensuring that all reachable code is executed.

By using the Branch coverage method, you can also measure the fraction of independent code segments. It also helps you to find out which sections of code don't have any branches.

By using the Branch coverage method, you can also measure the fraction of independent code segments. It also helps you to find out the sections of code that don't have any branches[7].

Formula

Branch Coverage=(No of executed branches/Total no of branches)*100

Tool: Jacoco

Metric 3: Mutation Score

Mutation Testing is a type of software testing where we mutate (change) certain statements in the source code and check if the test cases are able to find the errors. It is a type of White Box Testing which is mainly used for Unit Testing[8].

Each mutated version is called a mutant and tests detect and reject mutants by causing the behaviour of the original version to differ from the mutant. The number of mutants depends on the definition of mutation operators and the syntax/structure of the software 100% mutation score means killing all mutants (or a random sample).

Formula

Mutation Score=(Killed Mutants/ Total number of Mutants) * 100

Tools: PITclipse and PIT Mutation Idea Plugin

Metric 4: McCabe Complexity (Cyclomatic Complexity)

M McCabe complexity is a software metric used to measure the complexity of a program. It is a quantitative measure of independent paths in the source code of the program. It uses the Control Flow Graph to calculate the complexity of the source code.

An independent path is defined as a path that has at least one edge which has not been traversed before in any other path. Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program.

Formulas

A) Cyclomatic Complexity = $E - N + 2P$, where

E = the number of edges in CFG

N = the number of nodes in CFG

P = the number of connected components in CFG

D = is the number of control predicate (or decision) statements

B) $D + 1$, where

D = is the number of control predicate (or decision) statements

Tool: JaCoCo

Metric 5: Code Churns

In Code Churn, we try to find out the number of lines changed to achieve the desired functionality. In brief is defined as lines added, modified or deleted to a file from one version to another. When code churn is suddenly higher than expected, it can be an early indicator that something is off with your team. Detecting code churn early means you can help diagnose what's going on and have a conversation at exactly the right time.

Formula

Total lines modified = (Total lines added in new version + Total lines modified).

Code Churn = Total lines modified / Total number of lines(LOC) in new version.

Tool: CLOC

Metric 6: Code Smells

Code smells are a set of common signs which indicate that code is not good enough and it needs refactoring to finally have a clean code. There are various code smells such as :

- **Type Checking:** Type-checking code is introduced in order to select a variation of an algorithm that should be executed, depending on the value of an attribute. Mainly it manifests itself as complicated conditional statements that make the code difficult to understand and maintain[9].
- **God Class:** The God object is a part of the code smell group and it is a kind of object that knows too much or does too much. That means a huge class in terms of the number of lines of code.
- **Feature Envy:** Feature envy is a code smell describing when an object accesses fields of another object to execute some operation, instead of just telling the object what to do.
- **Long Method:** whenever a method contains many code lines, then it creates a code smell type called Long Method. New lines of codes are added to it regularly without anything being optimized.

Code smells are not bugs or errors. Instead, these are absolute violations of the fundamentals of developing software that decreases the quality of code. This increases the need for such issues to be addressed as early as possible.

Tool: Jdeodorant

IV. STEPS TO COLLECTING THE DATA

We divide the data collection process into the following steps:

- Step 1: Choosing the open-source projects (two open-source projects having at least 100K SLOC)
- Step 2: Building the project and running the test cases.
- Step 3: Configuring JaCoCo - Code Coverage Library for Java.
- Step 4: Installing/ Configuring PITclipse and PIT Mutation Idea Plugin for Mutation Testing.

Step 5: Installing CLOC for calculating Code Churn.

Step 6: Installing Eclipse Plugin Jdeodorant for calculating code smells.

Step 7: Generating and running the script for correlation calculation and visualization.

Step1: Choosing the open-source projects

A good amount of research was done for finding and selecting the open-source projects based on the following criteria:

- At Least two of the 4 open-source projects must be above 100K SLOC, all having majority of Java code.
- Each project should have at least 5 released versions of subversion.
- Each project should have enough Junit test cases in order to collect data for mutation testing.
- The chosen projects must have documentation that can guide towards successfully building the project.

Step 2: Building the projects and running the test cases.

After selecting various open-source projects, the next step is to build these projects. We faced many issues while building these projects due to the following reasons :

- These projects were using different build tools such as Maven, Ant etc so building these configurations according to individual system requirements was a bit challenging.
- Projects had build errors due to missing some dependencies, incompatible JDK versions, Junit incompatibilities.

Next, we ran test cases to make sure that we have enough test cases running to calculate statement coverage, branch coverage, mutation testing coverage in each of the projects.

Step 3: Configuring JaCoCo - Code coverage library for Java.

Metrics 1, 2 and 4 are about the statement coverage, branch coverage and Cyclomatic complexity. So, to get that data we chose JaCoCo library since it is a free and popular code coverage library for Java. Since we are using the IntelliJ Idea IDE, it was easy to configure the JaCoCo within IDE. IntelliJ Idea has built-in JaCoCo configured. Following are steps how JaCoCo is configured and some of the necessary criteria need to be followed to compute these metrics :

Step 1: Build the project without any compilation errors and make sure all tests are running successfully as shown in Fig 1.0.

Step 2: Create a new Run/Debug configuration. Go to Run > Create/Edit configuration. As shown in Figure 1-1, choose “All in package” as test kind. Also, make sure to choose the configuration for JaCoCo to run as shown in Figure 1-2.

Step 3: Apply the newly created run profile and run JaCoCo with test coverage. Go to Run > “Run JaCoCo with Coverage”.

Step 4: Export the generated coverage results as shown in Figure 1-3.

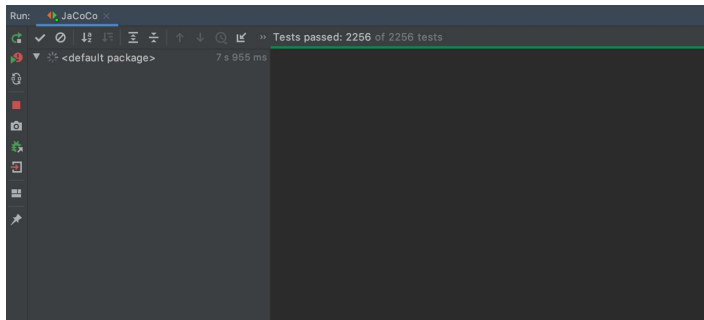


Figure 1-0: All tests are running as shown in figure.

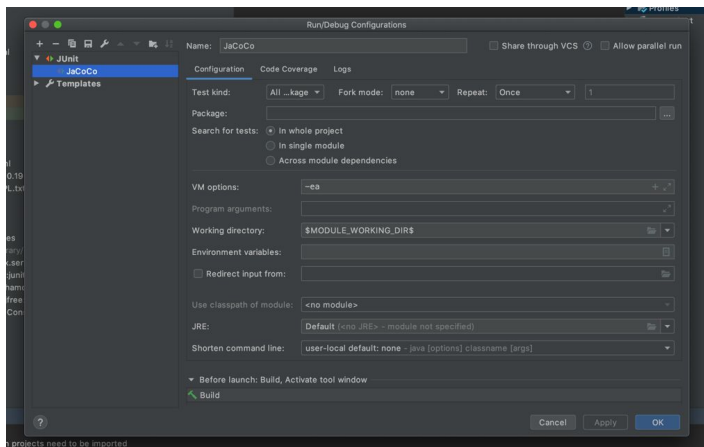


Figure 1-1: JaCoCo Configuration in IntelliJ Idea IDE

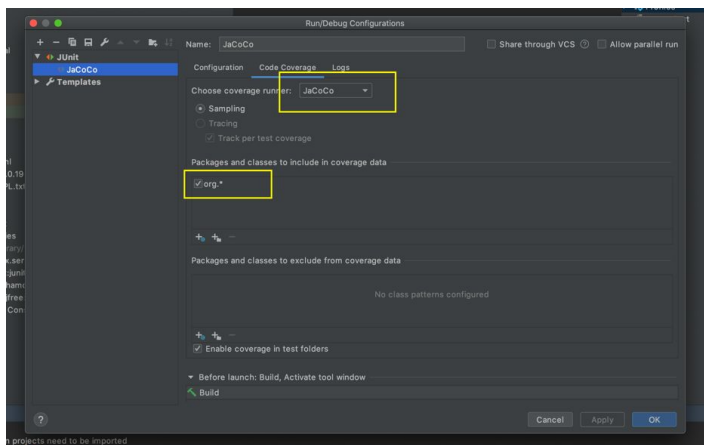


Figure 1-2: JaCoCo Configuration in IntelliJ Idea IDE

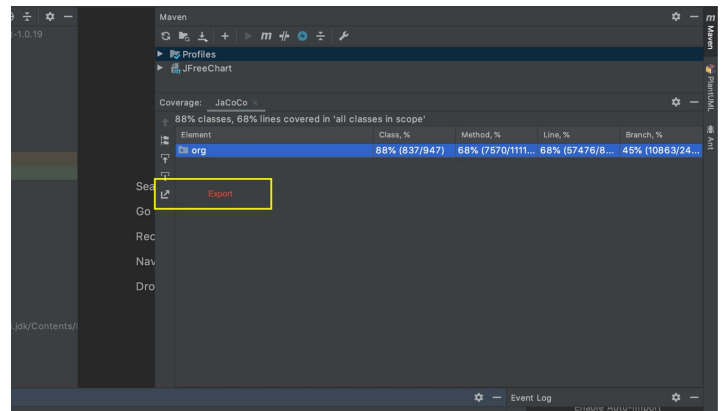


Figure 1-3: Export generated report.

ifreechart\$JaCoCo.exec

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cov	Missed	Lines	Missed	Methods	Missed	Classes
org.freechart.plot	75%	53%	2,102	4,240	3,907	14,861	690	1,911	5	79		
org.freechart.renderer.xy	53%	30%	1,386	2,158	3,225	7,605	228	834	7	73		
org.freechart.svg	71%	49%	1,439	2,717	2,583	9,192	278	1,135	3	88		
org.freechart.renderer.category	59%	35%	1,101	1,774	2,612	6,619	140	658	1	55		
org.freechart	68%	36%	861	1,489	1,847	5,380	316	804	11	73		
org.freechart.time	61%	49%	526	1,376	1,054	5,428	165	844	3	48		
org.freechart.editor	0%	0%	240	240	961	961	119	119	13	13		
org.freechart.util	25%	17%	275	327	767	1,034	106	141	5	19		
org.freechart.fx	0%	0%	287	287	809	809	159	159	4	4		
org.freechart.fx.demo	68%	52%	241	550	655	2,219	64	274	0	28		
org.freechart.general	74%	0%	19	19	417	417	18	18	3	3		
org.freechart.annotations	71%	51%	249	542	637	2,135	94	315	1	30		
org.freechart.xy	68%	64%	462	1,325	656	4,342	175	840	2	73		
org.freechart.time	67%	41%	286	536	482	1,671	73	257	0	19		
org.freechart.renderer	80%	53%	347	775	534	2,585	92	390	3	27		
org.freechart.fx.demo	73%	59%	258	550	552	2,128	142	357	0	26		
org.freechart.statistics	65%	60%	200	699	394	2,483	67	341	0	27		
org.freechart.labels	77%	50%	256	573	374	1,874	57	319	0	47		
org.freechart.fx.interaction	48%	37%	114	200	269	575	31	108	0	20		
org.freechart.category	74%	56%	124	279	310	1,002	32	144	0	8		
org.freechart.entity	67%	37%	158	266	278	776	78	186	1	22		
org.freechart.fx.interaction	0%	0%	95	95	263	263	42	42	8	8		
org.freechart	90%	72%	166	618	301	2,504	28	314	0	30		
org.freechart.config	0%	0%	77	77	185	185	31	31	2	2		
org.freechart.fx.demo	84%	60%	83	106	227	283	17	28	0	3		
org.freechart.data	0%	0%	19	19	164	164	18	18	3	3		
org.freechart.data.percent	0%	57%	116	285	173	1,002	58	171	0	10		
org.freechart.data	0%	60%	80	246	246	33	33	3	3	3		
org.freechart.data.xml	0%	0%	72	72	199	199	45	45	8	8		
org.freechart.data.xml	0%	53%	53	53	143	143	24	24	3	3		
org.freechart.encoder	0%	0%	41	41	107	107	38	38	5	5		

Figure 1-4 Sample JaCoCo report

Step 4: Configuring PIT for Mutation Testing.

Metric 3 provides the test suite effectiveness, and we chose the PIT tool (<http://pitest.org>) to calculate the mutation score. IntelliJ Idea has a plugin that simply provides results when we run mutation tests on the selected test suites. The tool adds a 'Run configuration' that allows executing PIT within IDE. Following are steps how Pitest is configured and some of the necessary criteria need to be followed to compute these metrics :

Step 1: Run->Edit Configurations->Defaults->Pit Runner or just simply right click on the project's parent directory and choose to run PITest from the context menu as shown in Figure 2-1. Note here, it will apply the mutation to all the classes in the project since we are choosing the same option.

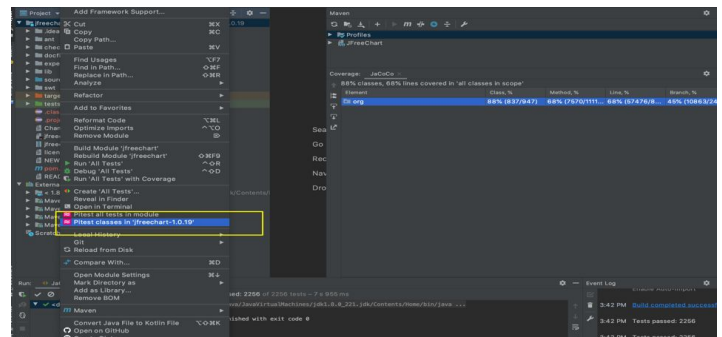


Figure 2-1 PIT Configuration.

Step 2: After completion of running the mutation testing, it will generate the test reports as shown in Figure 2-2.

Step 3: A script written in Java was created which takes two folder paths as input i) Path of Jacoco folder. ii) Path of PIT Report folder. This will generate the PIT csv files for corresponding folders.

Pit Test Coverage Report

Project Summary

Number of Classes 505 Line Coverage 54% 29577/54496 Mutation Coverage 32% 11326/35640

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
org.jfree.chart	21	49% 1801/3679	21% 463/2154
org.jfree.chart.annotations	16	58% 808/1395	31% 297/956
org.jfree.chart.axis	46	58% 3588/6130	30% 1316/4335
org.jfree.chart.block	16	66% 828/1260	42% 325/766
org.jfree.chart.demo	3	0% 0/164	0% 0/117
org.jfree.chart.editor	13	0% 0/963	0% 0/435
org.jfree.chart.encoders	5	0% 0/107	0% 0/47
org.jfree.chart.entity	13	43% 206/484	29% 76/266
org.jfree.chart.event	9	59% 47/80	7% 2/29
org.jfree.chart.fx	4	0% 0/809	0% 0/447
org.jfree.chart.fx.demo	3	0% 0/417	0% 0/271
org.jfree.chart.fx.interaction	8	0% 0/263	0% 0/139
org.jfree.chart.imagemap	5	52% 42/81	53% 20/38
org.jfree.chart.labels	26	63% 628/1002	53% 232/440
org.jfree.chart.needsle	10	31% 123/393	21% 67/320
org.jfree.chart.numerical	2	13% 34/261	6% 9/157
org.jfree.chart.plot	46	65% 7080/10896	30% 2134/7044
org.jfree.chart.plot.dial	12	59% 806/1357	32% 296/934
org.jfree.chart.renderer	11	68% 1110/1641	53% 582/1092
org.jfree.chart.renderer.category	24	46% 2119/44716	17% 527/3130
org.jfree.chart.renderer.xy	32	42% 2290/5439	20% 738/3639
org.jfree.chart.series	1	100% 3/3	100% 1/1

Figure 2-2 PIT test coverage report

Step 5: Code Churn using the CLOC library.

For analyzing the code churn we used the CLOC library which provides the functionality for the comparison between five versions of the project and it generates the result in the form of csv. The tool itself is very easy to use and have the following steps to use :

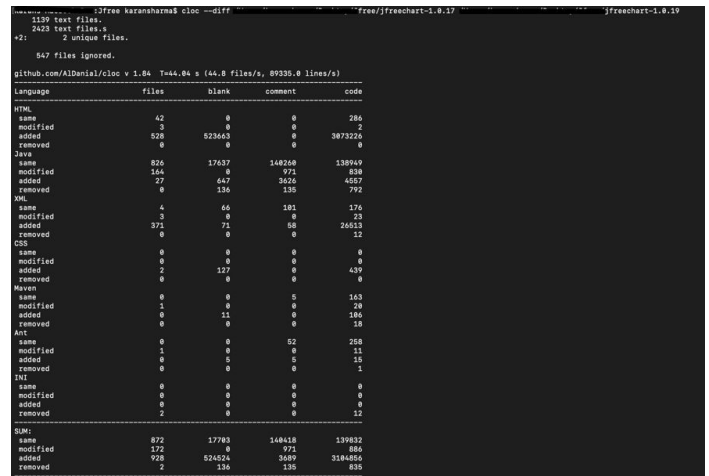
Step 1: Install CLOC using the following command :

```
npm install -g cloc
```

Step 2: Specify the older and newer versions of the project along with the output file to print out the results as shown in Fig 3.0 and results are shown in Fig 3.1.

```
cloc [options] --diff <set1> <set2>
Compute differences of physical lines of source code and comments
between any pairwise combination of directory names, archive
files or git commit hashes.
Example: cloc --diff Python-3.5.tar.xz python-3.6/
cloc --help shows full documentation on the options.
```

Figure 3.0 Cloc command usage



Language	Files	blank	comment	code
HTML	same 42	0	0	286
modified	3	0	0	2
added	528	523663	0	3873226
removed	0	0	0	0
Java	same 826	17637	148208	138949
modified	146	0	171	839
added	27	647	3626	4657
removed	0	136	135	792
XML	same 4	66	181	276
modified	3	0	0	23
added	371	71	68	24613
removed	0	0	0	0
CSS	same 0	0	0	0
modified	0	0	0	0
added	2	127	0	439
removed	0	0	0	0
Maven	same 0	0	5	163
modified	1	0	0	28
added	0	11	0	186
removed	0	0	0	18
Ant	same 0	0	52	268
modified	1	0	11	11
added	0	5	5	15
removed	0	0	0	1
INI	same 0	0	0	0
modified	0	0	0	0
added	0	0	0	0
removed	2	0	0	12
SUM	same 872	17783	148418	139832
modified	172	0	171	866
added	928	524524	3687	3184054
removed	2	136	135	835

Figure 3.1 Cloc generated report

Step 6: Analyse code smells and collects data for different versions of the software.

For the code smells, we basically used the Jdeodorant plugin available in Eclipse. We decided to select the latest five versions of the project. There are various code smells that are available in the code of these five versions as shown below. There are following steps that need to be followed to use this plugin:

Step 1: Install JDeodorant from Eclipse and restart Eclipse. See Fig 4.1.

Step 2: In the toolbar menu, select the Bad Smell menu as shown in Fig 4.2 and can use any kind of code smell we want to select.

Step 3: Then, select the package we want to find code smells in and click on the “i” button and it will show results and we can export results to file as shown in Fig 4.3.

Step 4: We repeated the same steps for all of the other code smells as well.

Step 5: The exported files are in .text format so we converted this data in csv format.

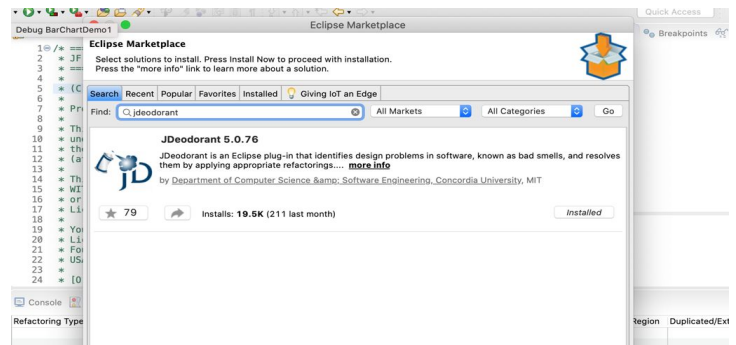


Figure 4-1: Jdeodorant Plugin

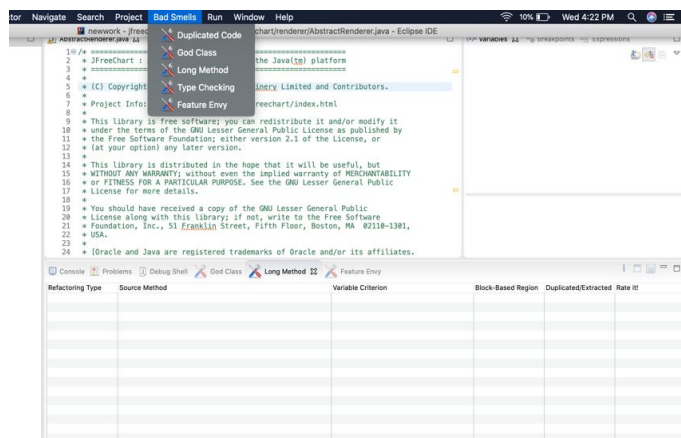


Figure 4-2: Options: Bad Smells

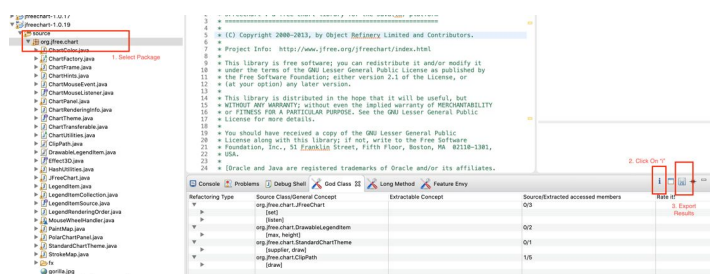


Figure 4-3: Generate Results

V. STEPS FOR ANALYZING THE DATA

We use the Spearman correlation coefficient to analyze correlation.

Steps of data analysis are as follows:

- Determining which two metrics are used for correlation comparative analysis and determining the level on which data need to be analyzed(e.g. Version level, Class level).
- Extracting the metric data of a specific project from the collected data.
- Importing the collected metric data into csv files.
- Running the python script and providing csv files as input to generate the value of the spearman coefficient and visualize the correlation.

Based on the correlation coefficient and graphs, we can derive a conclusion of an association between various metrics.

VI. CORRELATION ANALYSIS

1) Correlation between Metric 1&2 and Metric 3

Spearman's rank correlation coefficient between Metrics 1,2 and 3.

Project	Spearman's coefficient of metric 2&3	Spearman's coefficient of metric 1&3
JFreeChart	0.771452861	0.841792354

JFreeChart	0.771452861	0.841792354
Apache Commons Digester	0.4153	0.5705
Apache Commons Math	-0.040809542	0.649222905
Apache Commons Configuration	-0.098962483	0.760254022

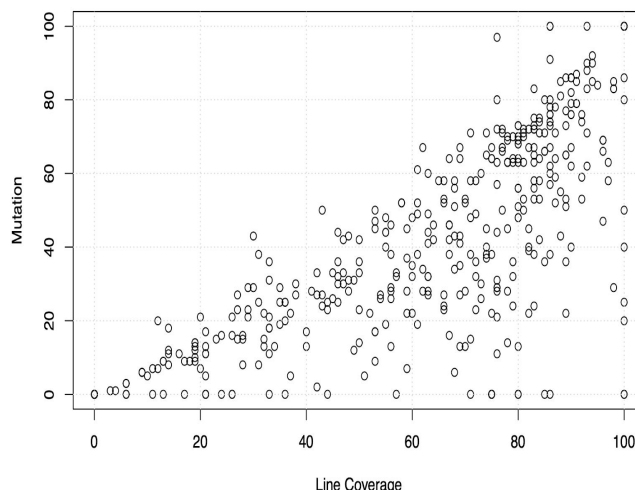


Fig: 1.1 - Project: JFree Chart

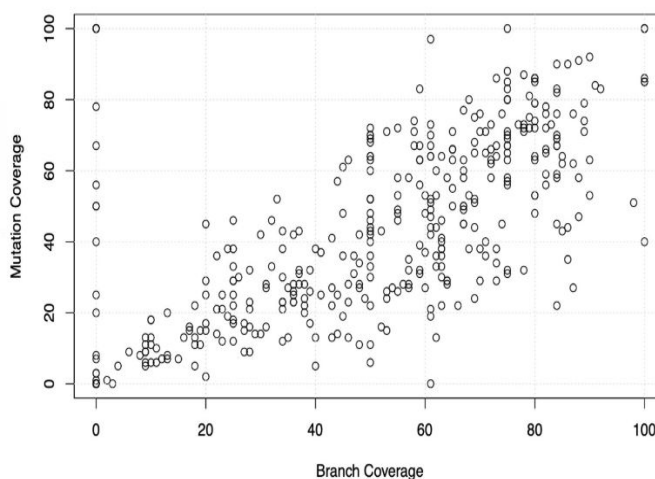


Fig: 1.2 - Project: JFree Chart

2) Correlation between Metric 1&2 and Metric 4

Spearman's rank correlation coefficient between Metrics 1,2 and 4.

Project	Spearman's coefficient of metric 1&4	Spearman's coefficient of metric 2&4
JFreeChart	0.945	0.613

Apache Commons Digester	0.946	0.851
Apache Commons Math	0.898	0.897
Apache Commons Configuration	0.926	0.893

Apache Commons Math	0.898	0.897
Apache Commons Configuration	0.09	0.09

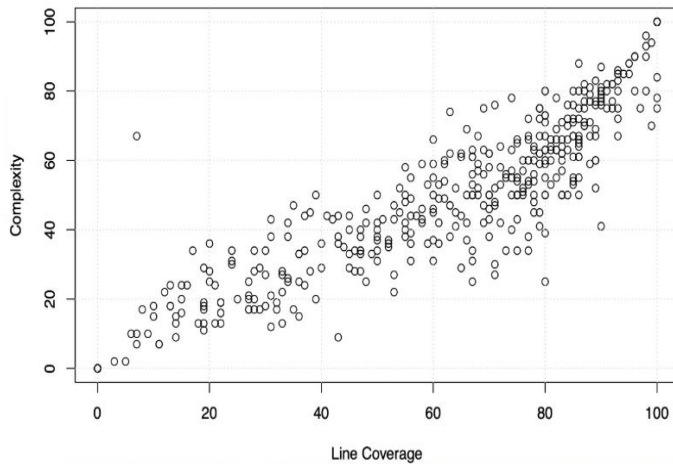


Fig. 2.1 - Project: JFree Chart

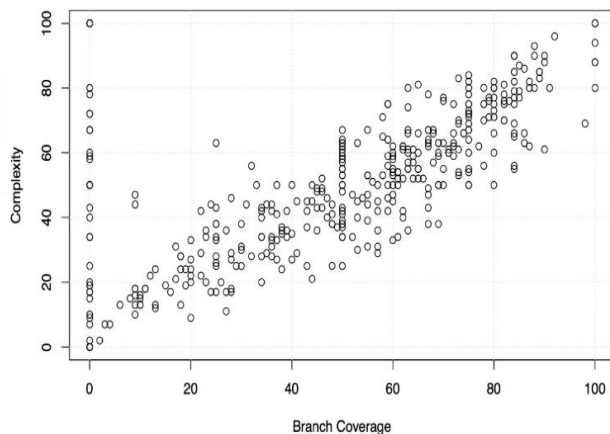


Fig. 2.2 - Project: JFree Chart

- 3) Correlation between Metric 1&2 and Metric 6
Spearman's rank correlation coefficient between Metrics 1,2 and 3.

Project	Spearman's coefficient of metric 1&6	Spearman's coefficient of metric 2&6
JFreeChart	-0.99	-0.782
Apache Commons Digester	0.9467	0.851

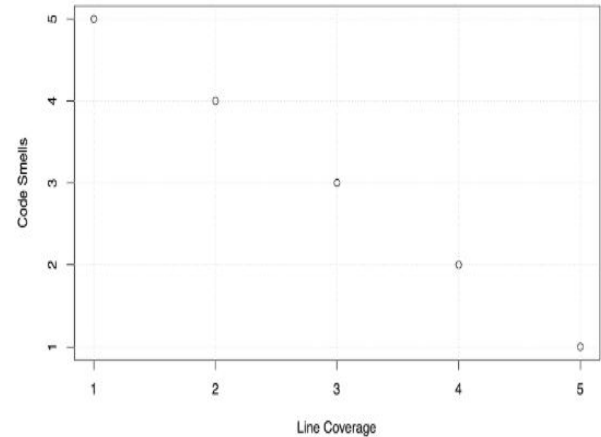


Fig. 3.1 - Project: JFree Chart

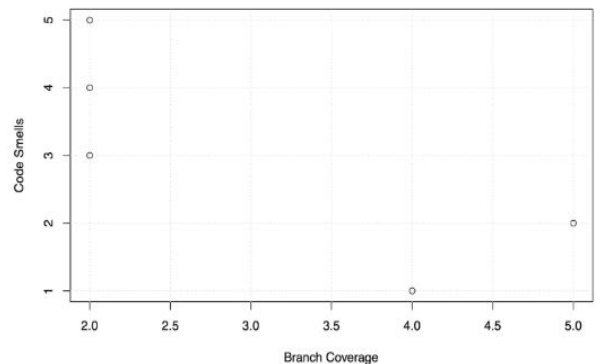


Fig. 3.2 - Project: JFree Chart

- 4) Correlation between Metric 5 and 6
Spearman's rank correlation coefficient between Metrics 5, 6.

Project	Spearman's coefficient of metric 5&6
JFreeChart	-1
Apache Commons Digester	-0.2
Apache Commons Math	-0.4
Apache Commons Configuration	-0.4

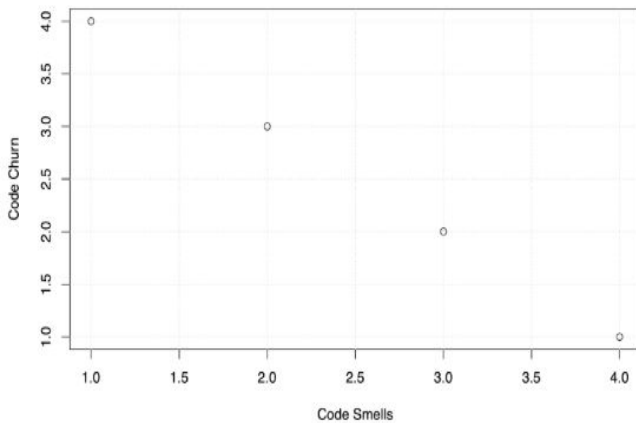


Fig: 4 - Project: JFree Chart

VII. CONCLUSIONS

Correlation between metric 1&2 and 3

The correlation between metric 1&2 and metric 3 is positive and very strong. We can conclude that suites with a higher statement or branch coverage show a high mutation score. This conclusion is consistent with the rationale that test suites with higher coverage can show better test suite effectiveness.

Correlation between metric 1&2 and 4

The correlation between metrics 1&2 and 4 is positive and the strength of the association is good but not very strong due to some inliers in branch coverage. We can conclude that classes with higher Cyclomatic Complexity show higher statement/branch coverage. This conclusion is consistent with the rationale that classes with higher complexity are more likely to have high coverage test suites.

Correlation between metric 1&2 and 6

The correlation coefficients for metric 1&2 and metric 6 are negative and small correlation. If the branch coverage and statement coverage increases, then Code Smells will decrease exponentially.

The rationale is that if the program has high branch coverage and statement coverage then it has a lower chance of containing undetected code smells. That means higher test coverage helps to increase software quality.

This conclusion is consistent with the rationale that classes with higher branch and statement coverage, then Code Smells will be less.

Correlation between metric 5 and 6

The correlation coefficients of the metric 5 and metric 6 are negative and small correlation. We can conclude if Code Churn increases, then Code Smells will decrease which means, if we get rid of most of the bugs from the previous version then the new version will have high software quality. This conclusion is

consistent with the rationale that if Code Churn increases then Code Smells decreases.

VIII. RELATED WORKS

- 1) Md Abdullah Al Mamun, Christian Berger & Jörgen Hansson [10] collected 24 code metrics classified into four categories, according to the measurement types of the metrics, from 11,874 software revisions (i.e., commits) of 21 open source projects. Our work is related to this work as we also collected data of 6 different metrics which belong to 6 different categories of metrics for 4 open source projects.
- 2) Tu Honglei, Sun Wei & Zhang Yanan [11] calculated McCabe methods and C&K metric methods for examples of complexity metrics to find a relation between complexity and other metrics. Our work is related to this work as we are calculating cyclomatic complexity using the Jacoco tool and then finding the correlation of cyclomatic metric complexity with Branch Coverage and Statement Coverage.
- 3) Yahya Tashtoush, Mohammed Al-Maolegi, Bassam Arkok [12] calculated metrics such as line of code, McCabe Complexity and found a correlation between them using the real dataset as a case study.

REFERENCES

1. https://www.researchgate.net/publication/321091593_Correlations_of_software_code_metrics_an_empirical_study
2. <https://github.com/apache/commons-math>
3. <https://commons.apache.org/proper/commons-configuration>
4. <http://commons.apache.org/proper/commons-digester/>
5. <http://www.jfree.org/jfreechart/>
6. <http://www.informit.com/articles/article.aspx?p=30306&seqNum=3>
7. <https://www.guru99.com/code-coverage.html>
8. <https://web.eecs.umich.edu/~weimerw/481/readings/mutation-testing.pdf>
9. https://users.ensc.concordia.ca/home/n/nikolaos/publications/CSMR_2008.pdf
10. <https://link.springer.com/article/10.1007/s10664-019-09714-9#auth-1> "Effects of measurements on correlations of software code metrics."
11. https://www.researchgate.net/publication/232617908_The_Research_on_Software_Metrics_and_Software_Complexity_Metrics "The Research on Software Metrics and Software Complexity Metrics"
12. https://www.researchgate.net/publication/264936702_The_Correlation_among_Software_Complexity_Metrics_with_Case_Study "The Correlation among Software Complexity Metrics with Case Study"