Software Measurement Project Report-Team F

Sukesh Kumar Mudrakola
Master of Software Engineering
Concordia University
Montreal, Canada
sukeshkumar.1995@gmail.com

Bhavana Bomma
Master of Software Engineering
Concordia University
Montreal, Canada
bomma08@gmail.com

Pruthvi Raju Nallaparaju
Master of Software Engineering
Concordia University
Montreal, Canada
npraju999@gmail.com

Birva Shah
Master of Software Engineering
Concordia University
Montreal, Canada
birvashah.shah3@gmail.com

Himani Patel
Master of Software Engineering
Concordia University
Montreal, Canada
himanip1996@gmail.com

Abstract— This document contains the information regarding the process we followed to calculate essential software metrics for 5 open source software projects. The document also contains the correlations derived among these metrics and the process followed to derive this correlation and all the essential calculations involved**.**

Keywords— *Code Coverage, Code Complexity, Projects, Mutation testing, Maintainability Index, Post-release Defect Density*

I. INTRODUCTION

In the contemporary world of IT, software has revolutionized the way things work. Almost all of the tasks which once needed a lot of human efforts for the accomplishment can be easily automated using software systems. With the growing importance and prominence of software in our daily lives, there is an absolute need for having a means of measuring the effectiveness. Software metrics are therefore tossed to make sure all the essential aspects of a software such as complexity, coverage, ability to take changes, bug fixing, and more can be measured.

Measurement lies at the heart of many systems that guides and governs our lives in terms of both personal and professional persona. We see measurement in almost everything around us. For instance, economic measurements determine price and pay. Measurements in flight radar systems enable us to detect aircrafts or airplanes when direct vision is obtained. In medical systems, measurements enable doctors to treat and diagnose specific illnesses based on the issue. Without measurement, technology cannot function. and there will practically be no means of knowing how effective and efficient the software system is. But measurement is not solely the domain of professional technologists. Each of us uses it in everyday life. Price acts as a measure of value of an item in a shop, and we calculate the total bill to make sure the shopkeeper gives us correct change. So, measurement helps us to understand our world by all means and helps us interact with our surroundings and improve our lives.

Various industry experts have found various ways to calculate these software metrics and among all the metrics available, there are few prominent metrics. In this paper, we have taken 6 such metrics to calculate how effective and efficient the choose projects are. Here are the metrics that will be discussed further in this paper.

1) Statement and branch coverage to calculate code coverage
2) PIT testing for testing the test suit effectiveness
3) Cyclomatic complexity using McCabe cyclomatic complexity to calculate project complexity
4) Maintainability index to measure the software maintenance effort
5) Post-release defect density to calculate the software quality

The paper will also demonstrate on how correlations between these metrics are formed. the hypothesis, the findings, and the conclusion. Correlation is a bivariate analysis that measures the relativity and strength of association between two variables and the direction of the relationship. We have implemented Spearman correlation to find and draw the correlation between these metrics.

II. PROJECTS SELECTED

A. *JFreeChart*

Size: 167,000LOC
Technologies underneath: Java
Project Link: https://www.openhub.net/p/jfreechart
Description: JFreeChart is an open-source java chart library, which makes it very easy and efficient for developers to display data in terms of graphs and charts. The tool supports various output types including Swing, JavaFX, image files, vector graphic file formats, and more. It is a class library for use of developers and is not an end-user application.

B. *Apache HttpComponents Client*

Size: 70,000LOC
Technologies underneath: Java
Project Link:
https://www.openhub.net/p/httpcomponents_client

Description: The Apache HttpComponents project is responsible for creating and maintaining a toolset of low-level Java components focused on HTTP and associated protocols. HttpClient on the other hand provides reusable components for client-side authentication, HTTP state management, and HTTP connection management.

C. *Apache Commons IO*

Size: 33,900LOC
Technologies underneath: Java
Project Link: https://www.openhub.net/p/commons_io
Description: Apache commons IO is written in java and provides a library for various utilities to develop input and outputs for software. It provides utility classes, input, output, filters, comparators, file monitor, and many more for easy development.

D. *Apache Commons Configurations*

Size: 847,000LOC
Technologies underneath: Java
Project Link: https://www.openhub.net/p/commons-configuration
Description: Apache Commons Configuration helps in providing a uniform source of having a common library to deal with various property and configuration files of various projects. It works by parsing all the configuration files such as properties, and more to get the right insights about a project.

E. *Apache Commons Collections*

Size: 132,000LOC
Technologies underneath: Java
Project Link: https://www.openhub.net/p/Apache-Commons-Collections
Description: Apache commons collections is a powerful framework which contains many comparator implementations, iterator implementation, adapter classes, utilities to test and create typical set-theory properties of collections such as union, intersection, closure, and more. It supports various powerful data structures that accelerate development of most significant java applications.

III. METRICS

A. *Statement Coverage*

The Statement coverage is also known as Segment Coverage or Line Coverage. Statement coverage is one of the white box testing which involves in identifying and checking number of statements in the source code. It is a metric, which is used to measure and calculate the number of executed statements in the program and which are not. It can also be used to check the quality of program. It covers only true conditions. This metric is always presented as a percentage. When the coverage is 100%, it means that every statement has been executed once.

coverage= (Number of statements executed/Total number of statements) *100%

In statement coverage testing, we make sure that every code block is executed. We can also identify which blocks are failed to execute. Normally for a company that uses statement coverage the typical coverage target is 80-90%, which means the outcome of the test should be such that 80-90% of the statements are executed at the end of the testing. Statement Coverage identifies those statements in any given method that have been performed. By doing so it can acknowledge the existence of those chunks or blocks of code that failed to perform their recognized task. Ultimately, identification of these blockages is the main job of Statement Coverage. The higher the Statement Coverage number the better is the quality of the written code. The best part of this testing technique is that it can be conducted by the code developers themselves.

B. *Branch Coverage*

The branch coverage is also known as All-edge coverage or Decision coverage. Branch coverage simply measures every branch in the program, each possible branch has been executed at least once. For example, if the outcomes are binary, you need to test both True and False outcomes. Branch coverage is simply checking a decision point and moving further accordingly, from one decision point to another, whichever relevant.

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

By using Branch coverage method, you can also measure the part of independent code sections. It also helps you to find out which segments of code don't have any branches. Branching is a jump from one decision point to another. Research is shown that even if functional testing is done, it achieves only 40/5-60% branch coverage. Branch coverage is stronger than that of statement coverage. Every outcome from a code module is tested. Branch coverage method removes issues which happen because of statement coverage testing and Allows you to find those areas which are not tested by other testing methods:

NOTE: 100% branch coverage guarantees 100% statement coverage, but 100% statement coverage does not guarantee 100% branch coverage.

Collecting and analyzing data

Calculating statement and branch coverage: The idea is simple. The application is started with a code coverage tool (JaCoCo) attached to it, then tests are executed, and results are gathered.

- Installation is done through **Eclipse -> Help -> Eclipse Marketplace… -> search for "jacoco" -> Install -> restart Eclipse.**
- Project have to be imported to Eclipse from **File** -> **Import** -> **Existing Maven projects.**
- After passing all the required tests, now Import the results into Eclipse **File -> Import -> Coverage Session -> enter name and select code.**
- Results can be exported, (Creates a code coverage report for tests of a single project in multiple formats (HTML, XML, and CSV)) using **File -> Export -> Coverage Report -> select session and location.**

Running a coverage analysis is as effortless as pressing a single button like the existing Run and Debug buttons. The coverage results are automatically summarized in the Coverage view and highlighted in the Java editors. 100% code coverage does not necessarily reflect effective testing, as it only reflects the amount of code exercised during tests. The calculated code coverage for all the projects chosen is depicted in Table-2 in Correlations Section.

C. *Test Suite Effectiveness*

Testing is an internal part of software development process used to improve the quality of software. It is crucial

to improve software tests because of increase in size and complexity of software. However, it is often difficult to do a complete testing as there are incompatibilities between the cost of testing and the number of faults it detects. Quality of a test suite often measures testing effectiveness. A test suite detecting more bugs directly implies the quality of that test suite.

One of the ways to measure the quality of test suite is through Mutation Testing. It initiates spurious defects to measure the quality of test suite. Mutation introduces defects by modifying a small piece of code that should result in an abnormal behavior when tests are run. If the tests fail, it can be inferred that some tests are not checking every possible behavior. So those tests need to be improved. Developers need to design new tests that can distinguish the behavior of mutants from the original program. There are two well-known tools available to calculate mutation score i.e Stryker and PIT testing. For our projects, we have chosen PIT, a practical mutation PIT testing tool for Java.

Why we have chosen PIT testing for Mutation Analysis?

As we have only considered Java Maven build projects for our study, it is likely to use PIT testing. Other mutation testing tools are comparatively slow, difficult to use, and are mainly build for academic research purpose. The objective of PIT is to provide a clear report indicating test execution combining line coverage and mutation coverage for each test suite. The advantage of using PIT is that it is fast, robust and easily integrable with development tools. We can also use command line interface to run PIT testing. We have used PITClipse plugin in Eclipse IDE to generate mutation score. As shown in Figure 1 and 2, it highlights the mutants that were killed and that were survived using different colors which is easy to understand. Also, it provides the list of mutation operators used to create mutants. PIT is publicly available at http://pitest.org/

Collecting and analyzing data

Tools used: PITClipse

Calculating Mutation Score: The mutation score is defined as the percentage of killed mutants with the total number of mutants.

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

Steps executed to generate a mutation score in PITClipse for all test suites:

1) Install PITClipse plugin in Eclipse

2) Run maven build project for PIT testing

3) Mutant generation and execution

4) Get Mutation Score from PIT Report generated in PITClipse console.

Steps executed to generate a mutation score in Command Line Interface for all test suites:

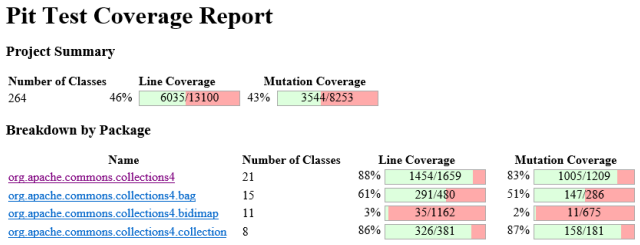
1) Add Pitest plugin in pom.xml file of project.

2) Run “maven clean install”

3) Run “mvn org.pitest:pitest-maven:mutationCoverage”

4) Mutant generation and execution

5) Get Mutation Score from PIT Report generated in target folder of project.



Pit Test Coverage Report

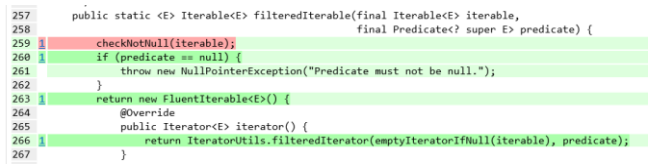
Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
264	46%	6035/13100	43%	3544/8253

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.apache.commons.collections4	21	88%	1454/1659	83%	1005/1209
org.apache.commons.collections4.bag	15	61%	291/480	51%	147/286
org.apache.commons.collections4.bidimap	11	3%	35/1162	2%	11/675
org.apache.commons.collections4.collection	8	86%	326/381	87%	158/181

Figure-1: Screenshot of PIT Report (It includes No. of Test Classes, Line Coverage and Mutation Coverage)



```
257     public static <E> Iterable<E> filteredIterable(final Iterable<E> iterable,
258                                                    final Predicate<? super E> predicate) {
259 1       checkNotNull(iterable);
260 1       if (predicate == null) {
261             throw new NullPointerException("Predicate must not be null.");
262         }
263 1       return new FluentIterable<E>() {
264             @Override
265             public Iterator<E> iterator() {
266 1               return IteratorUtils.filteredIterator(emptyIteratorIfNull(iterable), predicate);
267             }
```

Figure-2: Screenshot of PIT Report generated by PITClipse (Light Green color shows Line coverage and Dark Green color indicates killed mutants, Light Pink indicates no coverage and Dark Pink color indicates remaining mutants)

We generated PIT reports as csv files for all projects. Analyzing the data, we found that Line Coverage is calculated based on mutated line of code. If one test case does not execute mutated line, then that line is not considered in line coverage. PIT report also provides total no. of test cases, line coverage and mutation coverage considering the master suite of project. It is calculated by taking *average* of line and mutation coverage of all test suites. We found that line coverage and mutation coverage are independent of no. of test cases in test suites. They depend on quality of test case based on mutants killed. Figure-3 shows that line coverage and mutation score for all test suites in all projects are following moderate linearity.

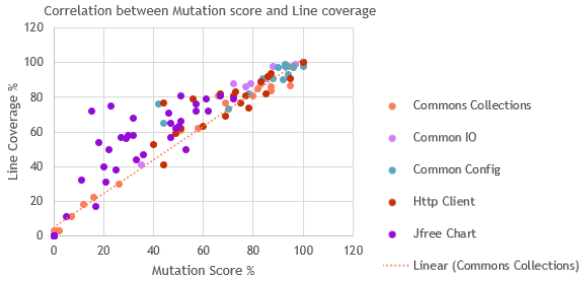


Figure-3: Correlation between line coverage and mutation score

The calculated mutation score % for all the projects chosen is depicted in Table-3 in Correlations Section.

D. McCabe Cyclomatic Complexity

Software risk and criticality are defined and are solely based on the complexity. About four decades ago, Thomas McCabe introduced the globally famous cyclomatic complexity (CC) Metric, which is still in very effective use and is used at an enterprise level. Even after 40 years, it is still one of the most globally used, popular, and powerful measurement technique to determine the complexity of the code.

Cyclomatic complexity is very simple to calculate and intuitive to understand. It can be taught quickly and can

be grasped easily to determine the complexity at class level depending on the conditional statements or linearly independent paths in a class. It can first be calculated class-wise and then can be taken to a package level or at a project level. Software complexity is a very crucial aspect in the field of software engineering. Software complexity refers to the complexity of the code developed and is based on various factors such as the number of looping or conditional statements, code complexity and more. McCabe essential complexity metric is also known as cyclomatic complexity and is the measure of depth and quantity of routines in a program/project.

McCabe complexity depends on the number of nodes, edges, and connected components in the control flow graph of the program. According to the McCabe, the cyclomatic complexity M is the measure of linearly independent paths for a program is. Which is:

McCabe Complexity = E - N + 2P.

where,
E-Number of edges in the graph
N-Number of Nodes in the graph
P-Number of connected components.

Collecting and analyzing data
Tools used: Sci-tools
Calculating McCabe Complexity: McCabe complexity can be calculated based on the number of edges, nodes, and the number of connected components at a class level. We have multiple tools which analysis projects in different ways to calculate the McCabe Complexity.

Some tools parse every project and calculate the number of branching conditions for keywords such as While, For, If, Else, Etc. Whereas, there are few powerful tools, which calculates the McCabe complexity for every class in the project directly. Amongst all the available tools, we have chosen Sci-Tools to calculate the McCabe's Cyclomatic Complexity. The rationale behind this is the reputation of the tool. Sci-tools is a globally used to calculate various software metrics. The tools parse the projects based on the file hierarchy and works in a very effective means. The tool also considers cluster cal graphs, function pointer support, Track floating license usage, and more, more information of the use of Sci-Tools and the working can be understood from the documentation available on their website. Users can refer (https://scitools.com/category/documentation/) to get more information on how sci-tools work.
The calculated average cyclomatic complexity for all the projects chosen is depicted in Table-2 in Correlations Section.

E. Maintainability Index

Software maintenance involves modification of the product to fix the bugs. This plays an important role in a way that software does not age. Maintainability is defined as the difficulty of changing a software system's source code, thus it is tied very closely to the concept of software maintenance. Maintainability Index was first proposed by Oman and Hagemeister and derived a formula of their own. This formula was modified and used in programs like Microsoft visual studio. After a lot of changes the final formula resulted:

MI = 171-5.2*ln(V) -0.23*(G) -16.2*ln (LOC)

Where the independent variables are

V- Halstead Volume

G- Cyclomatic complexity

LOC- Source lines of code (SLOC).

According to the formula if the maintainability index is greater than 45 indicates a class or a project indicates good maintainability. If it is less than 45 then it requires high maintenance. The individual parameters Halsted volume, cyclomatic complexity and SLOC involves changes in overall value. Halsted volume involves the number of operators and operands, if there is an increase in these values results in increase of Halstead volume which in turn decreases maintainability index that requires high maintenance. Similarly, higher the cyclomatic complexity involves higher the control predicates that results in less maintainability index.

Collecting and analyzing data
Tools used: Prest and Sci-tools
Calculating Maintainability Index: We have taken five different projects and calculated the maintainability index for each class and then found the average of all the classes to get the maintainability index of individual projects and their versions. In order to calculate the Maintainability index, we have used Prest jar file to calculate Halstead volume. **PREST** is easy to use as it just takes the input files and parse the results as methods, classes and packages. From these we took the class results. In order to calculate Cyclomatic complexity and source lines of code we used **Sci-tools** which takes all the project files and analyzes the data. The export metric provides an option to choose the required metric values that can be parsed in form of a csv file. The results from both the files are combined in a common Excel file and a macro was written using maintainability index formula to get the maintainability index for each class. The Maintainability Index of a project was calculated by taking the average of all the classes.

The calculated maintainability index for all the projects chosen is depicted in Table-5 in Correlations Section.

F. Post-Release Defect Density

The Post-Release Defect Density is a metric which can be compared across different versions of projects as well as across various projects. The post-release Defect Density is basically used to indicate the Product Quality. The total number of source lines of code are used by the calculation of the post-release Defect Density.

Defect Density=Defect count/size of the release (SLOC)

- Defect Count is the total number of post-release defects
- SLOC are the total number of source lines of code

Factors that affect the defect density metrics

- Code complexity
- The type of defects considered for the calculation
- Time duration which is considered for Defect density calculation
- Developer or Tester skills

Defects that are found in the production environment are also called the post-release defects (or operational defects). A defect is also referred to as bug as well as faults. It is also defined as a static fault in software e.g. incorrect lines of code. Post-release defects are generally identified after the delivery of the version of the project to the production environment. We can say that this version is an affected version for the particular bug or defects found.

Collecting and analyzing data

Tool Used: JIRA, BugZilla

Calculating Post-release Defect Density*:* Both these tools (JIRA and BugZilla) are very widely-used project trackers used for bug tracking, issue tracking, and efficient project management. To be able to uniformly obtain bug information for the different projects in our dataset, we searched for open-source projects that use JIRA or Bugzilla issue tracking system. It should also allow the issues that records in the issue tracking system.

Source line of code collection

First, we found projects as listed above and most of the projects are developed by the Apache Foundation except JFreeChart. The reason for that is Apache is well maintained for bug filings in a separate issue tracking system and also allow public access. Hence, it is easy to track the bug as well as getting all necessary details required for the calculation of the post release defect density metric. For each project, we have considered 6 versions for better analysis and statistics. Then, we collected the source code of projects that are hosted on GitHub and used JIRA issue tracking system. After collecting all 6 versions, for each one of them, we used Sci-tools to collect SLOC.

Bug collection at project level

In order to get the defects lists for each version of the individual project, we visit the project's official website and we search for the defects list and we found the link for the external issue tracking system that the specific project have used. We perform this step manually for each project. Now, for each bug, JIRA and BugZilla records all the necessary details including the affected version of the software.

We collected the bugs along with essential information such as bug ID, affected version, fix version, priority, product, component, status, and summary.

Analysis and Descriptive summary of the collected data and result

Project Name	Versions	No of Post-Release Defects	SLOC	Post-Release Defect Density
Apache Commons Io	2.4	64	24384	0.002624672
Apache Commons Configurations	1.1	39	15181	0.002569001
Apache Commons Collections	4.0	36	51608	0.000697566
Apache Commons Io	2.5	24	29152	0.000823271

Table-1: Analysis of collected data for Post-Release Defect Density Metric

Here we have considered top four highest no. of bugs from all selected projects. We have considered two data sets for analysis of this metric, **1) Post-Release Defect Density and 2) No. of Bugs.** Table-1 shows the post release defect density of 6 versions of each of 5 projects. As we have seen in the formula, the post release defect density is proportional to number of bugs and inversely proportional to the SLOC. As shown in the Table-1, the top 2 highest number of bugs among all this projects has been found in the version 2.4 of Apache Commons Io and version 1.1 of Apache Commons Configurations with the bugs of 64 and 39 respectively .Hence, its post release defect density values are also the top 2 highest among all the values. Now if we consider the next 2 highest value of the bugs which is in version 4 of Apache Commons Collections and in version 2.5 of Apache Commons Io with the bugs of 36 and 24 respectively and with the SLOC of 51608 and 29152. But here you can see the post release defect density value is higher in the version 2.5 of Apache Commons Io which has the lesser bugs compared to the 36 bugs in the version 4 of Commons Collections. This is because the SLOC of version 2.5 is far lesser than version 4 of Commons Collections So that, it depicts that it also inversely depends on the SLOC. Also,the latest version of the each project has the least number of bugs. This is because the most stable version of the each project is the latest version with fixes of the all post release defects that has been found in the previous versions.

We had started our study with the assumption that no. of bugs increases with SLOC. However, our study did not prove it strongly. For some projects, it clearly shows that bugs increase with increase in size while its different scenario in other projects. This shows that there is no exact trend between no. of bugs and SLOC. The calculated post release defect density for all the projects chosen is depicted in Table-5 in Correlations Section.

IV. CORRELATIONS

A. Correlation between Statement and branch coverage with Cyclomatic complexity

We have started the correlation with a hypothesis that, the projects with higher complexity are less likely to have high code coverage.

However, proceeding further, we found that there might be other factors which affects the complexity. In order to calculate the correlation between the code coverage and cyclomatic complexity, we have gathered the necessary statement and branch coverage information from sci-tools, which has been elucidated in 'Tools and calculations used-Section V' section. Once, we had the necessary data, which is the code coverage related information as well as the average cyclomatic complexity, we have found the spearman correlation coefficient.

Sr. no	Project Name	Average Cyclomatic Complexity (1)	Statement Coverage (2)	Branch Coverage (3)	Spearmen Correlation (1&2)	Spearmen Correlation (1&3)
1	Apache Commons Collections	14	65%	49%	-0.278	0.469
2	Apache HTTPComponents Clients	11	69%	23%	0.0458	0.497
3	JFreeChart	24	72%	35%	-0.506	0.488
4	Apache Commons IO	16	90%	45%	-0.4	-0.0369
5	Apache Commons Configurations	18	86%	56%	-0.369	0.533

Table-2: depicts the code coverage as well as the cyclomatic complexity with the correlation coefficient.

B. Correlation between Statement and Branch coverage with Test suite effectiveness

We have started our analysis with a hypothesis that, Code coverage is strongly correlated to Mutation score and increases linearly with the increase in Mutation score. While choosing projects, we have taken into consideration the number of test suites implemented. More no. of test suites implies better analysis of effectiveness. Table-3 shows that out of five projects, Commons Collection and Commons IO have the highest value of Spearman correlation coefficient between Code Coverage and Mutation Score. We have shown *Spearman Correlation Coefficients* in Table-3, however it is not calculated based on code coverage% and mutation coverage% as seen in Table-3. To calculate *Spearman Correlation Coefficient*, we have taken data range having code coverage and mutation scores for all test suites in all projects. However, it is remaining unclear that the effectiveness is affected due to test suite size or coverage of the test suite. We found that as Code Coverage increases, Mutation Score also increases in most of the test suites of projects. But in JFreeChart, Mutation Score is not strongly increase with the coverage. This shows that sometimes Code coverage is moderately correlated to Mutation score.

Project	Code Coverage %	Mutation Score %	Spearman Correlation Coefficient
Commons Collection	65%	43%	0.962
Http Client	69%	53%	0.850
JFreeChart	72%	33%	0.869
Commons Configuration	86%	80%	0.868
Commons IO	90%	80%	0.958

Table-3: Showing sorted Code coverage, Mutation score, and Spearman Correlation Coefficient.

C. Correlation between Statement and branch coverage with Post-release defect density

We have started the correlation with a hypothesis that, the projects with low code coverage contains more bugs.
Statement and branch coverage gives an estimate of the number of source code of lines executed. Therefore, if more coverage is achieved, then there are less chances for bugs to creep in to the system. However, there were some projects, which did not hold true for the hypothesis. Turns out, there might be other factors which affects the number of bugs. In fact, a better code coverage implies less errors while development.

However, post-release defect density considers the bugs after a software version is released. Therefore, factors such as testing tools, experience of testers, and post-release defects more affects compare to code coverage.

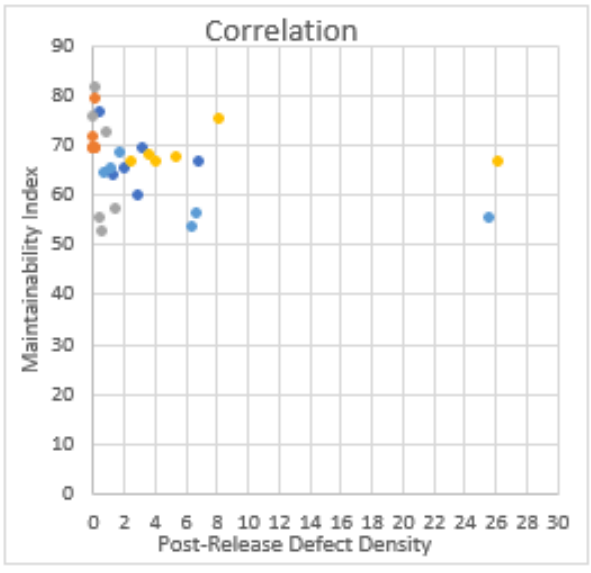
S. no	Project Name	Statement Coverage	Branch Coverage	Number of Bugs
1	Apache Commons Collections	65%	49%	3
2	Apache HTTPComponents Clients	69%	23%	2
3	JFreeChart	72%	35%	5
4	Apache Commons IO	90%	45%	24
5	Apache Commons Configurations	86%	56%	6

Table-4: Correlation between code coverage and number of bugs

We have achieved a strong Spearman correlation coefficient of 0.9(strong) when calculated against statement coverage and post-release defects. Moreover, we achieved 0.5 (medium) when calculated against branch coverage and post-release defects.

D. Correlation between Post-release defect density and maintainability index

We have started the correlation with a hypothesis that, with higher maintainability index we achieve less number of post-release defects.
Maintainability index reflects the ease of maintaining a project. That means with higher maintainability index, the cost and effort needed to make changes or fix bugs is reduced. Having said that, having a better/higher maintainability index ensures less software maintenance costs. However, it might not always ensure a bug free system. Therefore, for few of our projects, the number of post-release defects had a very weak correlation with that of the maintainability index as shown in Table-5.



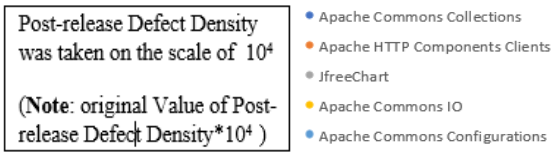


Figure-4: Scatterplot for correlation between Maintainability Index and Post release defect density

Figure-4 shows correlation between Maintainability Index and Post-release defect density. It shows most of the values of maintainability index span between 50 and 80. Moreover, the values of post release defect density span between 0 and 8 on the scale of 10^4.

Project Name	Versions	No of Post Release Defects	SLOC	Post-Release Defect Density	Maintainability Index	Spearman Correlation Coefficient
Apache Commons Collections	1	1	27083	3.692E-05	63.82	0.143
	2	4	13563	0.0002949	59.4	
	3	15	45670	0.0003284	69.03	
	4	36	51608	0.0006976	66.52	
	4.1	13	60717	0.0002141	65.01	
	4.2	3	62645	4.789E-05	76.3	
Apache HTTP Components Clients	4.4.1	1	74293	1.35E-05	71.42	-0.46
	4.5.2	2	75021	2.67E-05	68.91	
	4.5.3	2	75427	2.65E-05	69.12	
	4.5.5	1	76095	1.31E-05	69.27	
	4.5.6	1	76130	1.31E-05	69.26	
	4.5.7	2	76128	2.63E-05	79.1	
JfreeChart	1.0.12	9	144406	6.23E-05	52.29	-0.49
	1.0.13	7	145897	4.80E-05	55.23	
	1.0.14	23	145030	0.0001586	56.86	
	1.0.15	13	144690	8.98E-05	72.24	
	1.0.16	1	144192	6.94E-06	75.52	
	1.0.17	3	144340	2.08E-05	81.29	
Apache Commons IO	2	8	21434	0.0003732	67.61	0.26
	2.1	12	22129	0.0005423	67.07	
	2.2	6	23789	0.0002522	66.28	
	2.3	10	23983	0.000417	66.5	
	2.4	64	24384	0.0026247	66.3	
	2.5	24	29152	0.0008233	74.94	
Apache Commons Configurations	1.1	39	15181	0.002569	54.97	-0.6
	1.2	12	18489	0.000649	53.34	
	1.3	19	27819	0.000683	56.04	
	2.1	12	65558	0.000183	68.35	
	2.2	8	66697	0.0001199	65.09	
	2.3	6	67096	8.942E-05	64.15	

Table-5: Correlation between Maintainability Index and Post-release Defect Density of the projects according to their versions.

Result on observations of correlations

- Higher code coverage might not necessarily mean that the project will have less bugs. Instead, coverage ensures there are no/less errors.
- Having a better/higher maintainability index ensures less software maintenance costs. However, it might not always ensure a bug free system.
- Higher code coverage ensures the efficiency of the project and has a correlation with complexity. However, the effectiveness of test suite used for coverage has a role to play.
- Testing the effectiveness of the test suite largely depends on expertise test professionals, knowledge of the test suite developers as well as the technologies and test automation used.
- The latest version of each project has the least number of bugs among the multiple versions of the same project since the latest version is the version with the fixes of all the bugs that has been previously found in the previous versions.
- It is evident from our correlation analysis that post release defect density is directly proportional to number of bugs whereas, inversely proportional to SLOC.

VII. TOOLS AND CALCULATIONS USED

A. JaCoCo

JaCoCo is a free code coverage library for Java, which has been created by the EclEmma team. It is a popular Java code coverage tool that can create coverage reports using a technique known as Bytecode instrumentation, which modifies the bytecode as it is stacked into memory. EclEmma is a free Java code coverage tool for Eclipse, available under the Eclipse Public License. It brings code coverage analysis directly into the Eclipse workbench. EclEmma records which parts of Java code are executed during a program launch. This technique is called code coverage analysis and typically used with automated testing like JUnit unit tests.

B. SciTools

We have used Sci-Tools to extract SLOC, Cyclomatic complexity, and other information. Below are the steps followed to extract the needed data, which we have used in our analysis.

Steps to extract project metrics from Sci-Tools

-Open the application, after you have downloaded and installed it in the system
-Click on **File->New project**
-Give the name of the project and load the directory path to the project which needs to be parsed and calculated metrics for.
-Click on **next** and select the programming language, which is Java in this case.
-Click on **Add directory** tab and add the directory path of the project and click on **OK**
-Click on **Project-> analyze** all files.
-Click on **Metrics->export metrics** and select the relevant metrics.
-Give the output file format and the destination and click on **Export** to get the data needed.

C. Calculating Spearman Correlation

Spearman's Rank correlation coefficient is one of the most-prominent technique which can be used to find out the strength and correlation between two variables.

Method used to calculate the Spearman correlation

- Create a table from your data and get the ordered pairs of two variables.
- Rank the two data sets. Ranking is achieved by giving the ranking '1' to the biggest number in a

column, '2' to the second biggest value and so on. The smallest value in the column will get the lowest ranking. This should be done for both sets of measurements or the variables used to find the correlation for.

- Tied scores are given the mean (average) rank.
- Find the difference in the ranks (d).
- Square the differences (d^2) To remove negative values and then sum them
- Calculate the coefficient (R_s) using the formula mentioned below.

When written in mathematical notation the Spearman Rank formula looks like this:

$$\rho = 1 - \frac{6\sum d_i^2}{n(n^2-1)}$$

Here,
ρ= Spearman rank correlation
di= the difference between the ranks of corresponding variables
n= number of observations

D. PIT Testing Tool: PITClipse

PIT stands for Performance Improvement Team. Basic idea of PIT testing tool is to run unit tests against automatically modified versions of application source code. It should produce different results and cause the unit tests to fail when there is even a slight change. If a unit test pass in this situation, it is said that test suite is not effective. We have used PITClipse plugin available in Eclipse as PIT testing tool as we are comfortable to use Eclipse for Java Projects.

V. Related Work

- We have calculated various metrics to find the correlation among them for the projects chosen. Every metric has a different means to calculate and there were existing set of tools available to measure these metrics.

- JaCoCo-based EclEmma, which is an eclipse plugin has been used to calculate the code coverage for the projects. The tool gave a detailed analysis on the results at the very granular class level. The tool also allowed us to export the reports in the desired formats including a HTML report for visual ease and CSV files for calculation ease.

- Pitclipse, which is yet another powerful Eclipse plugin has been used to calculate the mutation scores for the test suits. Just like EclEmma, this tool also provides data in both HTML as well as CSV files.

- We have used the very traditional way to calculate McCabe Cyclomatic complexity, and maintainability index which depends on the number of linearly independent paths in a program. We have the results class wise, thanks to the tools (EclEmma, and Sci-Tools) which made our job easy.

- Post release defect density is based on the number of bugs found in the system. We have calculated that on a version-wise basis, of whose' source code has been extracted from Git repos.

VIII. Summary and concluding remarks

We have considered 5 different projects to find various correlations among the chosen metrics. After careful consideration and based on our analysis, we believe, it would be safe to conclude that a particular metric cannot give an absolute estimate about the project. Instead, metrics give valuable insights, however, there might be various other aspects to consider such as the size of the project, knowledge of the persons performing the tests, coverage, number of bugs, interval between the project release, and many more. All the independent findings and results of the correlations we made have already been stated in the '*results and observations*' section of Correlation analysis.

However, metrics do give a reliable means of measuring the software in their own respective manner.

IX. References

[1] G. K.Gill and C. F. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *IEEE Trans. Software Eng,* vol. 17, no. 12, Dec 1991.

[2] H. Liu, X. Gong, L. Liao and B. Li, "Evaluate How Cyclomatic Complexity Changes in the Context of Software Evolution," *42nd IEEE International Conference on Computer Software & Applications,* 2018.

[3] A. Kaur, K. K. and K. Pathak, "A Proposed New Model for Maintainability Index of Open Source Software," *Proceedings of 3rd International Conference on Reliability, Infocom Technologies and Optimization.*

[4] C. Cuiule, "Predicting Maintainability for Software Applications Early in the Life Cycle," *Price Systems.*

[5] P. S. Kochhar, F. Thung and D. Lo, "Code Coverage and Test Suite Effectiveness: Empirical Study with Real Bugs in Large Systems," *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering,* 2015.

[6] P. V. Beckhoven, A. Oprescu and M. Bruntink, "Assessing Test Suite E↵ectiveness Using Static," *Software Improvement Group,* vol. 2070.

[7] H. Coles, T. Laurent, C. Henard, M. Papadakis and A. Ventresque, "PIT a Practical Mutation Testing Tool for Java (Demo)," 2016.

[8] M. Alenezi, A. Hussien, M. Akour and M. Z. Al-Saad, "Test Suite Effectiveness: An Indicator for Open Source Software Quality," 2016.

[9] L. Vinke, "Estimate the post-release Defect Density based on the Test Level Quality," 2011.

[10] D. Gable, G. Rothermel, S. Elbaum, "The impact of software evolution on code coverage," *Proceedings of International Conference on Software Maintenance,* 2001.