Software Measurement Project Report-Team F

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

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

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

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

Pruthvi Raju Nallaparaju  
Master of Software Engineering  
*Concordia University*  
Montreal, Canada  
npraju999@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.  
-Statement and branch coverage to calculate code coverage.  
-PIT testing for testing the test suit effectiveness.  
-Cyclomatic complexity using McCabe cyclomatic complexity to calculate project complexity.  
-Maintainability index to measure the software maintenance effort.  
-Post-release defect density to calculate the software quality.

The paper will also demonstrate on how correlations between these aforementioned 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**

## **JFreeChart**

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.

Size: 167,000LOC  
Technologies underneath: Java   
Project Link: <https://www.openhub.net/p/jfreechart>

## **Apache HttpComponents Client**

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.

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

## **Apache Commons IO**

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.

Size: 33,900LOC  
Project Link: <https://www.openhub.net/p/commons_io>

## **Apache Commons Configurations**

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.

Size: 847,000LOC  
Technologies underneath: Java  
Project Link: <https://www.openhub.net/p/commons-configuration>

## **Apache Commons Collections**

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.

Size: 132,000LOC  
Technologies underneath: Java  
Project Link: <https://www.openhub.net/p/Apache-Commons-Collections>

**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.

**Advantages:**

#### Identifies which block of code is not executed.

#### Robust against plain code formatting.

#### Checks the flow of different paths in the program

**Disadvantages:**

#### It cannot measure false conditions

#### Without test cases the statement cannot be executed

#### Doesn’t report the termination conditions.

**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:

## **Advantages:**

#### Helps to ensure that no branch leads to abnormal behavior of the application.

#### Validate all the branches

#### It allows you to find a quantitative measure of code coverage

## **Dis-advantage:**

Ignores branches in Boolean expressions.

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

**Calculating statement and branch coverage**

**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 particular program launch. This technique is called code coverage analysis and typically used with automated testing like JUnit unit tests.

## **Methodology:**

The idea is pretty simple. The application is started with a code coverage tool (JaCoCo) is 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, nowImport 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 reflects effective testing, as it only reflects the amount of code exercised during tests.

# **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 way 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 has to be improved. Developers need to design new tests that can distinguish the behavior of mutants from the original program.

There are two tools available to calculate mutation score i.e Stryker and PIT testing. For our project, we have chosen PIT, a practical mutation PIT testing tool for Java. The objective of PIT is to provide a clear report indicating test execution for each test suite. It also highlights the mutants that were killed and that were survived using different colors. The advantage of using PIT is that it is fast, robust and easily integrable with development tools. We can also use command line interface. It also optimizes execution of mutants. We have used PITClipse plugin in Eclipse IDE to generate mutation score. However, we have cross checked the mutation score using command line interface too. PIT is publicly available at <http://pitest.org/>

**Methodology:**

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

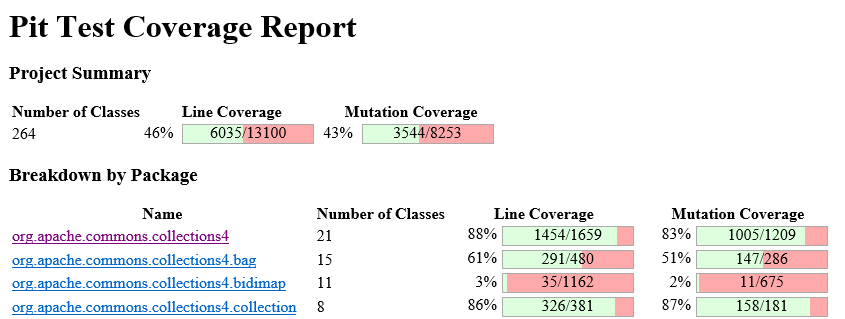
Steps executed to generate a mutation score at package level as follows:

1) Installing PITClipse plugin in Eclipse

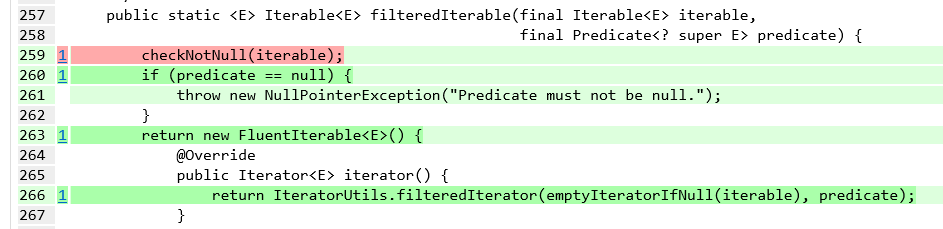
2) Running maven build project for PIT testing

3) Mutant generation and execution

4) Mutation Report



Example output of PIT(1)



# **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 as a whole.

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.

**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 Ski-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.

# **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.

**Calculating Maintainability Index**

**Tools used: Prest and Sci-tools**

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 codewe 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.

# **F. Post-Release Defect Density**

The post-release Defect Density is a metric which we can compare across the different versions of the projects as well as 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**

Defects that are found in the production environment are called the post-release defects (or operational defects)

Let set Di = {d1, ..., dn} where

• Defect d is a flaw in a component or system that can cause the component or system to fail to perform its associated function. 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. A defect, if encountered during execution, may cause a failure of the component or system. Post-release defects are generally identified after the delivery of the particular version of the project to the production environment.

So that, In other words we can say that Post release defects are basically the bug which is introduced after the delivery of the particular version. We can say this particular version as the affected versions for the particular bug or defects found.

**Calculating Post-release Defect Density**

**Tool Used: JIRA, BugZilla**

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 focus on projects that use JIRA/BugZilla for reporting bugs.

##### ***Source code collection***

##### First, we search for open-source projects that use JIRA or BugZilla issue tracking system and allow public access to all of the issues that records in the tracking system. We find various projects as listed above and most of the projects are developed by the Apache Foundation except Jfreechart. The reason for that is Apache well maintained the filing of bugs in a separate issue tracking system and allow the public access. So that, it is very easy to track the bugs as well as getting the all necessary details required for the calculation of the post release defect density metric. For each project, we have considered the 6 versions of each project for better analysis and statistics. We, then, collect the source code of projects that are hosted on GitHub and use JIRA issue tracking. After collecting the all the 6 versions, for each one of them, we use Scitools for these projects to collect LOC

##### ***Bug collection (Project Level)***

In order to get the defects lists for the 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. As we mentioned earlier, we have selected the projects that use the issue tracking system as JIRA or BugZilla. We perform this step manually for each software project Now, For each bug, JIRA and BugZilla records the affected version of the software.

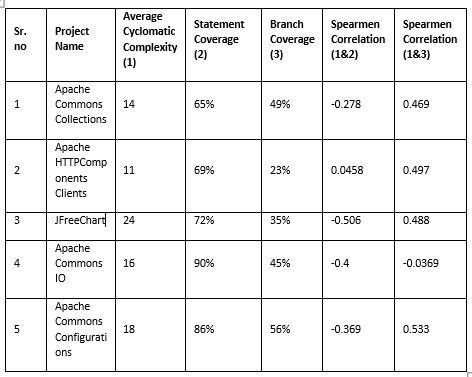
We collected the bugs along with essential information such as bug ID, affected version, fix version, priority, product, component, status, and summary.

##### **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.



**Table (1) depicts the code coverage as well as the cyclomatic complexity with the correlation coefficient.**

##### **B. Correlation between Statement and branch coverage with Mutation score**

We have started the correlation with a hypothesis that, Code coverage is strongly correlated to Mutation score and increases linearly with the increase in Mutation score. We have taken such projects that have large number of test suites implemented. Table-1 shows that from five projects, Commons Collection and Commons IO have the strongest Correlation between Code Coverage and Mutation Score. However, it is still not clear that the correlation with effectiveness was due to test suite size or coverage of the test suite. So, we have taken top five test suites having highest no. of test cases for each project to check correlation between Code Coverage and Mutation Score as shown in Table 2. 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.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Suites | No. of Test Cases | Code Coverage | Mutation Score |
| org.apache.commons.collections4.functors | 52 | 85% | 82% |
| org.apache.commons.collections4.iterators | 43 | 84% | 87% |
| org.apache.commons.collections4.map | 36 | 30% | 26% |
| org.apache.commons.collections4 | 21 | 88% | 83% |
| org.apache.commons.collections4.set | 16 | 62% | 58% |
| org.apache.commons.configuration2 | 37 | 91% | 84% |
| org.apache.commons.configuration2.tree | 25 | 99% | 93% |
| org.apache.commons.configuration2.builder | 21 | 98% | 94% |
| org.apache.commons.configuration2.io | 15 | 82% | 78% |
| org.apache.commons.configuration2.builder.combined | 13 | 98% | 93% |
| org.apache.commons.io.input | 30 | 86% | 77% |
| org.apache.commons.io.filefilter | 23 | 98% | 88% |
| org.apache.commons.io | 19 | 91% | 85% |
| org.apache.commons.io.output | 18 | 88% | 72% |
| org.apache.commons.io.comparator | 10 | 99% | 97% |
| org.apache.hc.client5.http.impl.classic | 32 | 62% | 51% |
| org.apache.hc.client5.http.impl.async | 26 | 0% | 0% |
| org.apache.hc.client5.http.impl | 18 | 53% | 40% |
| org.apache.hc.client5.http.impl.auth | 18 | 69% | 69% |
| org.apache.hc.client5.http.impl.cookie | 18 | 77% | 75% |
| org.jfree.data.xy | 39 | 72% | 57% |
| org.jfree.chart.axis | 37 | 57% | 27% |
| org.jfree.chart.plot | 34 | 68% | 32% |
| org.jfree.chart.renderer.xy | 31 | 40% | 20% |
| org.jfree.chart.util | 26 | 47% | 36% |

Table 2: indicates Code Coverage and Mutation Score for top five test suites having highest no. of test cases for each project.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Project | Code Coverage | Mutation Score | Number of Mutants | Number of Survived Mutants | Spearman Correlation Coefficient |
| Commons Collection | 65% | 43% | 8253 | 4709 | 0.962 |
| Commons IO | 69% | 53% | 3727 | 729 | 0.958 |
| Commons Configuration | 72% | 33% | 6156 | 1210 | 0.868 |
| JFreeChart | 90% | 80% | 34245 | 22850 | 0.869 |
| Http Client | 86% | 80% | 5032 | 2353 | 0.850 |

Table 3: Showing Code coverage, Mutation score, Number of mutants, Survived mutants and Spearman Correlation.

##### **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 os 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 more affects post-release defects on top of 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: Provides correlation between 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.

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.

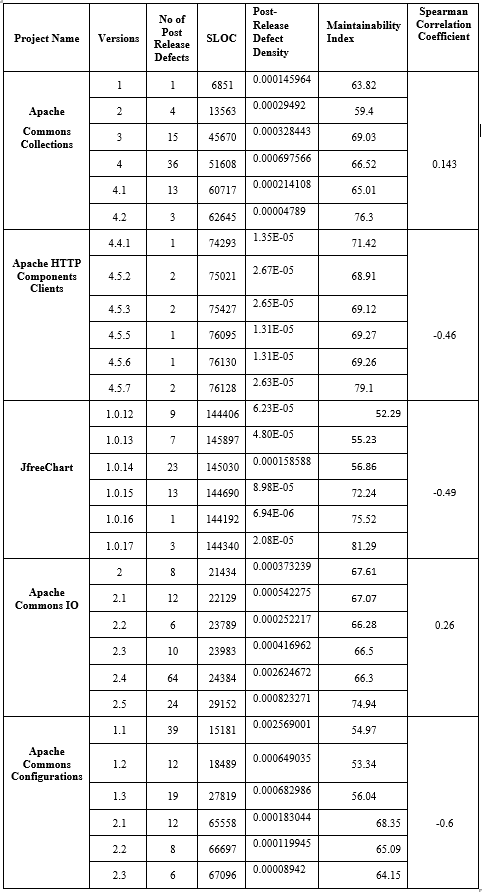
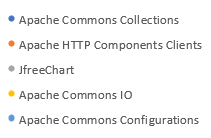
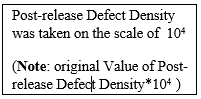


Table 5: Correlation between Maintainability Index and post release defect density of the projects according to their versions.

Scatterplot for correlation between Maintainability Index and Post release defect density.



##### **V Tools and calculations used**

**A. 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.

**B.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 (d²) To remove negative values and then sum them
* Calculate the coefficient (***Rs***) using the formula mentioned below.

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



Here,

ρ= Spearman rank correlation

di= the difference between the ranks of corresponding variables

n= number of observations

##### **Acknowledgment *(Heading 5)***

##### **References**

*[1] Geoffrey K.Gill and Chris F. Kemerer “Cyclomatic complexity density and software maintenance productivity,” IEEE Trans. Software Eng., Vol 17, No. 12, Dec 1991*

*[2] Huihui Liu, Xufang Gong, Li Liao, Bixin Li “Evaluate How Cyclomatic Complexity Changes in the Context of Software Evolution,” 2018 42nd IEEE International Conference on Computer Software & Applications.*

*[3] Arvinder Kaur, Kamaldeep kaur, Kaushal Pathak “A Proposed New Model for Maintainability Index of Open Source Software”* [*Proceedings of 3rd International Conference on Reliability, Infocom Technologies and Optimization*](https://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7000954)

*[4] Cara cuiule PRICE systems “Predicting Maintainability for Software Applications Early in the Life Cycle”*