

**SOEN 6611**

**Software Measurement**

**Project Report**

**By**

**Team - N**

|  |  |  |
| --- | --- | --- |
| Full Name | Student ID | Email |
| Venkata Pavan Kumar Reddy Ravi | 40083392 | pavan.03121996@gmail.com |
| Swetha Chenna | 40092019 | swethachenna2018@gmail.com |
| Nandini Bandlamudi | 40105415 | nandu.angel555@gmail.com |
| Gurminder Pal Dhiman | 40076840 | gurminderpaldhiman@gmail.com |

Abstract:

To identify the relationship between variables we used correlation analysis of metrics using spearman correlation coefficient and to analyze quality of software we considered six metrics and three open source projects to perform calculation , based on the results we estimated the impact on system

Keywords:

Statement Coverage , Branch Coverage, Mutation Testing , McCabe Complexity, Maintainability Index, Post Release Defect Density.

Introduction:

In the never ending evolution of software world, quality is considered as most important aspect where the automation takes over the prominence in current technological environment and the need to measure the system effectiveness has increased .A good software with no bugs is considered as efficient and to achieve this every project team needs to evaluate and correlate every factor of software.

Measurement is not just limited to software but to everything in our everyday lives. Best example will be a medical system where measurement is mandatory to identify the type of disease or to measure the price of an item in supermarts. Measuring quality can improve software development lifecycle and helps the accuracy of software. In this paper we as a team worked on three different open source maven projects where two of them are above 100k SLOC and one is below 100k SLOC that are developed using java programming language

Numerous ways are available all over the internet to measure the software metrics but according to subject expertise there are few effective metrics to calculate. In this we will discuss projects we selected and six metrics we calculated which is followed by correlation analysis that measures association between variables and paper also includes findings.

The metrics to calculate code coverage used are statement and branch coverage ,for test suite effectiveness we chose mutation coverage and to test we used PIT Test coverage ,to calculate complexity of project we used McCabe cyclomatic complexity ,to measure maintenance effort we used Maintainability index, to calculate quality we used Post-release defect density.

Open Source Projects Selected:

1.Apache Commons Configurations:

Size: 122k

Technology: Java

Source: <https://github.com/apache/commons-configuration/releases/tag/CONFIGURATION_2_0>

Description: Commons configuration 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. We worked on different versions.

2.Apache Commons Collections:

Size:118 KLOC

Technology:Java

Source: <https://github.com/apache/commons-collections>

Description: The java collections Framework was a major addition in JDK 1.2.It added many powerful data structures that accelerate development of most significant java applications. Since that time it has become the recognized standard for collection handling in java. Commons-Collections seek to build upon the JDK classes by providing new interfaces, implementations and utilities. Features include bag interface, comparator implementations, iterator implementations, Adapter classes from arrays and enumerations. We worked on different versions.

3.Apache Commons DBUTils:

Size: 15k

Technology: Java

Source: <https://github.com/apache/commons-dbutils/releases/tag/DBUTILS_1_7>

Description: The Commons DbUtils library is a small set of classes designed to make working with JDBC easier. JDBC resource cleanup code is mundane, error prone work so these classes abstract out all of the cleanup tasks from your code leaving you with what you really wanted to do with JDBC in the first place: query and update data.

Metrics:

1. Statement Coverage:

To indicate thoroughness of software quality testing we use test coverage which is measured as statement coverage where count refers to the number of statements executed at least once.If every statement is executed then we consider it as 100% coverage .This helps in verifying if source code is fulfilling required actions.

Formula: Number of statements executed/Total number of statements in source code \*100

Data Collection and Analysis:

Tools used : JaCoCo

Reason why we chose JaCoCo is

To calculate statement coverage we used the JaCoCo EclEmma plugin in Eclipse. The steps involved in the process to measure code coverage are:

* Install EclEmma plugin in Eclipse through Eclipse>Help>Eclipse-MarketPlace>Search for EclEmma>Install and Restart eclipse.
* Import the maven project.
* Add JaCoCo dependency information in the pom.xml file of the maven project and run the project as Maven Clean Install.
* JaCoCo results will be generated in the target folder of the project.
* Results will be generated in HTML, CSV and XML formats.

2. Branch Coverage:

To cover the defects in statement coverage we use branch coverage where cryptic errors in source code are detected. In this the number of branches executed in control flow graph at least once is considered which leads to finding existing bugs .If the full coverage is achieved then it will not allow errors where requirements are not met and validating all branches in source code to make sure there is no abnormal behavior.

Formula :Number of Decision Statements Executed/Total Number of Decision Outcomes\*100

Data Collection and Analysis:

Tools used : JaCoCo

To calculate statement coverage we used the JaCoCo EclEmma plugin in Eclipse. The steps involved in the process to measure code coverage are:

* Install EclEmma plugin in Eclipse through Eclipse>Help>Eclipse-MarketPlace>Search for EclEmma>Install and Restart eclipse.
* Import the maven project.
* Add JaCoCo dependency information in the pom.xml file of the maven project and run the project as Maven Clean Install.
* JaCoCo results will be generated in the target folder of the project.
* Results will be generated in HTML, CSV and XML formats.

3. Test Suite Effectiveness:

We chose mutation testing which is a white box technique through which we can obtain mutation scores. It is primarily used as a program based technique which uses operations to mutate the program and generate mutants and so we will create minute modifications of the program and generate mutants of original source code. Here the motive is to identify and kill mutants .When outcome is same then mutant is present if not killed.

Formula: Killed mutants/Total number of non-equivalent mutants

Data Collection and Analysis:

To calculate Pitest-mutation Maven-Pitest plugin has been used. The steps involved are:

* Import the Maven project into Eclipse.
* Edit pom.xml by adding the plugin information of Maven-Pitest plugin in the build>plugins section.
* Run the program as Maven Clean Install.
* MutationCoverage of the project will be generated to the target folder of the project once the project is successfully built.
* Data is generated in HTML and CSV formats.

Steps:

4. McCabe Complexity:

Cyclomatic complexity is referred to as an indicator for modularization ,to revise specifications , test coverage and software quality predictive models. We use this to predict fault numbers through multivariate regression analysis and find code with chances of errors. It is measured based on a linearly independent path that has at least one edge that has not traversed to any paths in the graph and is represented in a single number. It can be calculated to functions, modules, methods or classes.

Formula :E-N+2

Where E=number of edges in graph

N = Number of nodes in graph

Data Collection and Analysis:

Tools Used: JaCoCo

To calculate statement coverage we used the JaCoCo EclEmma plugin in Eclipse. The steps involved in the process to measure code coverage are:

* Install EclEmma plugin in Eclipse through Eclipse>Help>Eclipse-MarketPlace>Search for EclEmma>Install and Restart eclipse.
* Import the maven project.
* Add JaCoCo dependency information in the pom.xml file of the maven project and run the project as Maven Clean Install.
* JaCoCo results will be generated in the target folder of the project.
* Results will be generated in HTML, CSV and XML formats.
* Consider complexity of both missed and covered branches while calculating McCabe complexity from JaCoCo report.

5. Maintainability Index:

The evolution of software involves maintenance of product and the need to fix bugs which keeps software from aging. This can be considered based on halstead volume which involves number of operators and operands if the value is higher then it decreases maintainability index which requires high maintenance, cyclomatic complexity where if complexity is higher then it involves higher control predicates which result in maintainability index , LOC.It is project level and measure maintainability of project. If the maintainability index is greater than 45 then it is considered as a project with good maintenance and if it is less than 45 then the project requires high maintenance.

Calculation :MI=171-(5.2\*ln(V)+0.23\*(G)+16.2\*ln(LOC))

V = Halstead Volume

G = Cyclomatic Complexity

LOC = Count of source lines of code

Data Collection and Analysis:

Tools used:Jhawk

We used the JHawk tool to calculate the maintainability index of the projects. Steps involved are:

* Download the JHawk tool from the internet.
* Load the project into the JHawk tool.
* Click analyze and run the project in the tool.
* Results will be extracted method wise, class wise and project wise.

|  |  |  |
| --- | --- | --- |
| Projects | Versions | Maintainability Index |
| Apache Commons DbUtils | 1.1  1.2  1.4  1.6  1.7 | 72.63  70.57  78.43  78.79  79.03 |
| Apache Commons Configurations | 1.8  2.0  2.2  2.4  2.6 | 70.41  68.92  72.6  73.92  75.16 |
| Apache Commons Collections | 2.0  3.0  3.2  4.1  4.4 | 79.62  78.96  75.24  60.56  61.06 |

6.Post Release Defect Density:

To maintain quality of a project and satisfy user requirements is a prime objective and when the project has multiple iterations then the challenges will be more when compared to new projects. To avoid such scenarios we chose post release defect density which measures quality for every unit identified after each release. It measures the defects relative to software size expressed as lines of code or function point and also requires waiting for the defect to be deployed and needs to be fixed in immediate release. The chance of increase in defects count in new versions is possible as the fixed defects might have extended defect possibility. There are different factors that affect defect density metrics like code complexity ,type of defects considered, developer or tester skills.

Formula: Number of defects/Size of release SLOC

Data Collection and analysis:

Tools used: JIRA and LocMetric

To collect bugs for each version we used JIRA where we found bug reports for all versions and for SLOC we used LocMetric tool to get Source lines of code .Based on which we can calculate defect count by number of active and closed issues .For SLOC we downloaded each version and when it is imported in LocMetric and when we click on Count Loc option it gives us the count. And for defect count we have gone through apache commons official website where we searched for defects list which gave us the issue tracking system for each project .It includes a list of bugs with summary, id, priority, status ,affected version , fixed version which is considered integral information for calculation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Projects | Version | SLOC(Source lines of code) | Number of Bugs | Post Release Defect Density |
| Apache Commons DbUTils | 1.4 | 8558 | 2 | 0.000514 |
| Apache Commons Collections | 4.1 | 118204 | 16 | 0.000263 |
| Apache Commons Configurations | 2.6 | 126123 | 4 | 0.0000592 |

**Correlation Analysis:**

***Calculating Spearman Correlation***

Spearman’s Rank correlation coefficient is one of the most-prominent techniques 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,

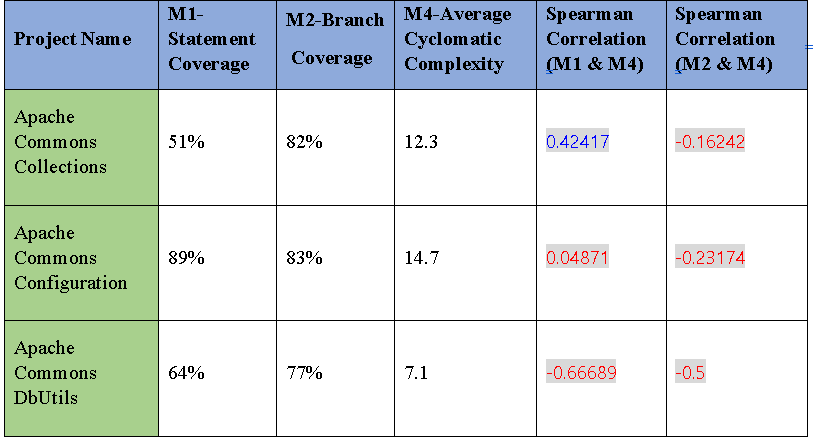
ρ= Spearman rank correlation

di= the difference between the ranks of corresponding variables

n= number of observations

**Correlation between Statement and Branch coverage with McCabe complexity**

We have started the correlation with a hypothesis that the project with higher complexity will likely have less code coverage.



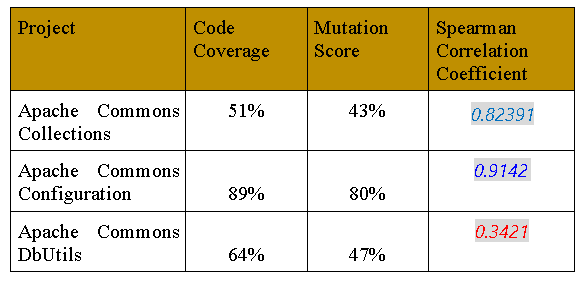
Negative correlation between branch coverage and cyclomatic complexity is observed.

Correlation between statement coverage and cyclomatic complexity is not strong. Positive values for Collections and Configuration is observed while DbUtils has negative correlation.

Correlation between Statement and Branch Coverage with McCabe Complexity is observed to be not strongly related. We found that there might be other factors affecting the complexity.

**Correlation between Statement and Branch coverage with Test suite effectiveness**

We have started the correlation with a hypothesis that code coverage is strongly correlated to mutation score and increases linearly with increase in mutation score.



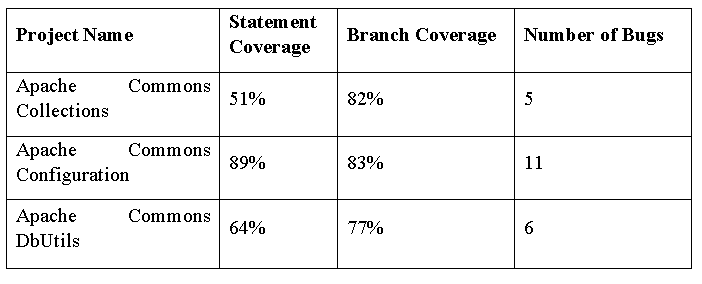
In Apache Commons Collections and Configurations, high Spearman coefficients are observed. Moderate Spearman’s coefficient value is observed in DbUtils. Correlation between Statement and Branch Coverage with Test Suite Effectiveness is observed to be not strongly correlated, that is with the increase in statement coverage doesn’t necessarily ensure high test suite effectiveness.

While choosing projects, we have taken into consideration the number of test suites implemented. More no. of test suites implies better analysis of effectiveness. 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 Apache Commons Collections, Mutation Score is not strongly increased with the coverage. This shows that sometimes Code coverage is moderately correlated to Mutation score.

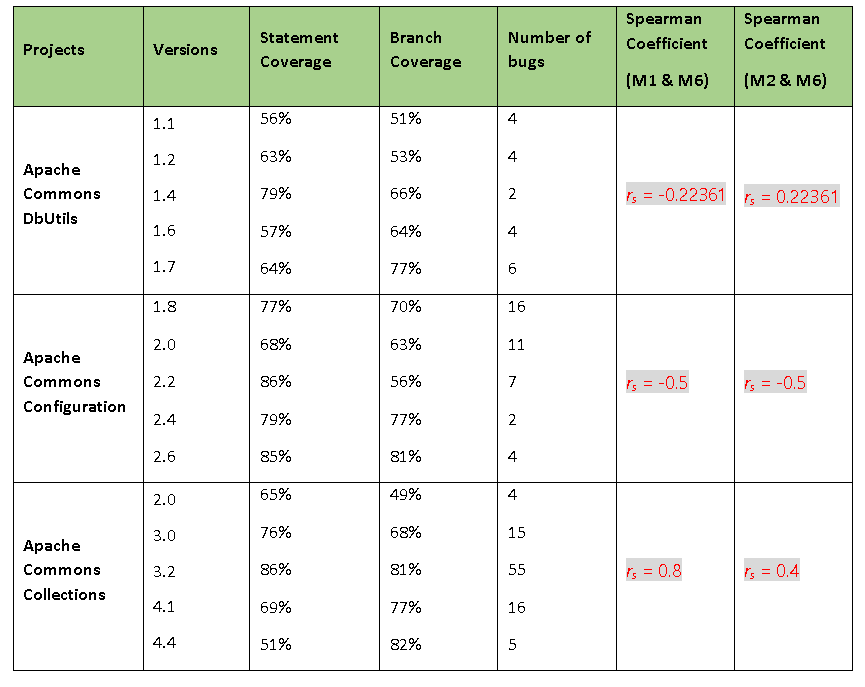
**Correlation between Statement and Branch Coverage with Post release defect density**

We have started the correlation with a hypothesis that the project with low code coverage will contain more bugs.

Code coverage gives us an idea of the thoroughness of testing by providing information about the amount of code that is tested. So that increase in code coverage is likely to lead to a decrease in post-release bugs.

****

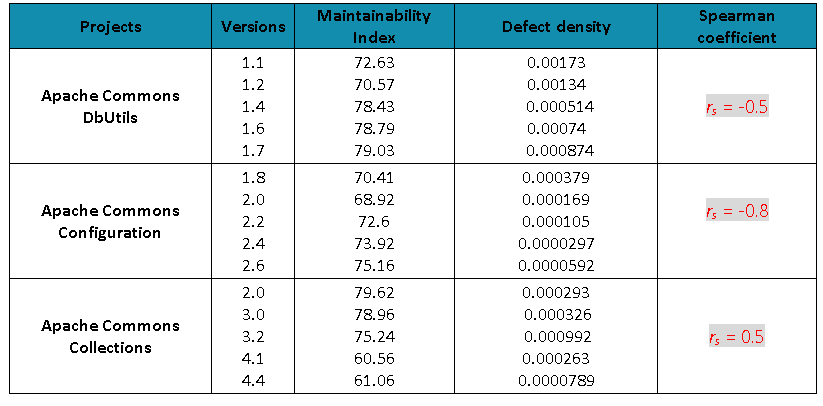
**The spearman coefficient for statement coverage and post release defect density is 1 i.e. strong correlation is observed from all projects. The spearman coefficient for branch coverage and post release defect density is 0.5 i.e. medium correlation is observed.**

****

For collections, both correlation values are positive. For configurations, both correlation values are negative.

**Correlation between Maintainability Index and Post release defect density**

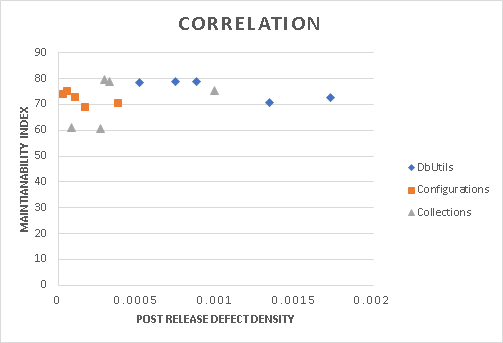
We have started the correlation with a hypothesis that with a high maintainability index we will have low post release defect density value.



Correlation for all the projects is between weak to medium.

From the analysis of correlation, the higher maintainability index indicates reduction in cost and effort needed to fix bugs i.e less maintenance cost. But, it might not ensure a bug free system. In addition, there are also some other factors that affect the value of the post release defect density like the experience of the developers and testers , the type of defects taken into account and the time required for the calculation of the post release defect density calculation.

**Scatter plot forPost Release Defect Density and Maintainability Index**



**Related Works:**

* We used different tools to calculate values for metrics and based on these results we performed correlation analysis.
* For statement coverage, branch coverage and cyclomatic complexity metric measures, we used Jacoco tool which is based on EclEmma. This is an eclipse plugin which gives results in both CSV and HTML format. By using this tool, we got detailed results at class level.
* To measure mutation score for test suites, we used PITest maven plugin through which we got results in HTMLformat.
* We used the Jhawk tool to measure maintainability index which gives results in method, class and project wise.
* To calculate post release defect density, we obtained the total count of bugs for each version from JIRA and to calculate SLOC, we LocMetric tool which is very easy to use.

**Conclusion:**

We have considered three different open source projects to analyse with respect to six different metrics. Based on our analysis, we found that analyzing and estimating a project needs more than one metric. There are other factors that also need to be considered like size, number of bugs and also the experience of resources that conducts the analysis and their capability to test. The complete analysis of correlation between metrics has been mentioned in previous sections.

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. 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.
3. L. Vinke, "Estimate the post-release Defect Density based on the Test Level Quality," 2011.
4. Relating Code Coverage, Mutation Score and Test Suite Reducibility to Defect Density https://ieeexplore.ieee.org/document/7528960 [7] Sharma, R. (2014).
5. Variations of Maintainability Values formula
   1. <http://www.virtualmachinery.com/sidebar4.htm>
6. M. Moghadam and S. Babamir, "Mutation score evaluation in terms of object-oriented metrics", 2014 4th International Conference on Computer and Knowledge Engineering (ICCKE), 2014. Available: 10.1109/iccke.2014.6993419
7. Spearman’s Coefficient: <https://geographyfieldwork.com/SpearmansRank.htm>