SOEN 6611: Software Measurement

Winter 2019

Project Report

Submitted to: Prof Jinqiu Yang

TEAM K		
Heer Oza	40043132	heeroza.ho@gmail.com
Meet Maniar	40039203	meetrmaniar@gmail.com
Shriyans Athalye	40037637	shriyansathalye@gmail.com
GitHub Link: https://github.com/shriyans16/SOEN_6611		

ABSTRACT

There are many available methods to measure overall software defect density, mutations score, statement coverage by the test suite. Some of the common ways help in identifying early defects, help maintain software from the results. This report represents some of the ways in which software defect density, mutation score, overall cyclomatic complexity, test suite effectiveness and maintainability index were calculated using different related metrics and analyzed to provide some correlation between each. For this project 5 different versions of 3 open source java projects were selected and recorded.

Keywords: Maintainability Index, GitHub, Cyclomatic Complexity, Defect Density, Mutation Score.

1.INTRODUCTION

Software in general consist of many ways for maintenance such as defect density, test suite effectiveness. This keeps the overall software under good condition and avoids conflicts in future revisions. Some of the software companies use automated tools for measuring these metrics while some do it manually. The major times these software metrics are calculated earlier has benefitted for the company to maintain their products better. This report consists of some of the known types of metrics measure done on open source projects to correlate if what metrics might affect on the other metric. This correlation was performed in many different research papers which were used to study in the project. Some of the projects measured the effectiveness of test suite [1] to compare if the test suites covered all the statements and measured it effectively whereas other research covered the maintainability of software. This measure helped in understanding by what way software projects are maintained on daily basis. In this project, projects with lines of code greater than 100k were considered to gather better result for comparison. For this project 5 versions with more than 1 year of release date were considered to study how it affects after every release year. This will help in measuring difference between the project of a longer period and calculate the difference. The projects were available from GitHub version wise. After data collection, other section analyses the correlation between selected metrics using spearman's correlation and measures if the collected metric affects each other inversely or not. This way it will help in analyzing whether metric collected matches with the generated hypothesis and are feasible at industry level. The following section defines and identifies the collection procedure and metric definition. Another section explains the procedure and analysis of the collected data and matches collected data with required metric correlation.

2. RELATED WORK

In this project there are many observations made during data collection and analysis. Some of them were correlated from previous projects. Mutation testing measure the percentage of killed mutants over the total number of mutants. Research [2] studied mutation score evaluation with terms of object-oriented metrics. The proposed study measured some of the object-oriented metrics which reflected the complexity of features that mutation operators are applied to them in program. Another research measured maintainability index on entropy-based approach on object-oriented metric. Many researches done until now focused mainly on object-oriented measures and analyzed results based on that. The difference between previously conducted research and our project were done on multiple projects, some having more than 100K lines of code or more than 3 projects whereas ours is restricted to only 3 projects with 5 versions. Of the selected ones Jfreechart[5] is a library for chart visualization used as a plugin in different projects . Apache common IO [6] is a library of utilities which assist in developing many of the IO functionalities whereas Apache Common codec [6] provides implementations of common encoders and decoders such as Base64, Hex, Phonetic and URLs.

Code coverage also being one of metric used for calculating test suite effectiveness measures overall coverage of statements and branch done in test cases written. This considers conditional statements being measured, number of statements covered in a test case to understand if all the functionality is being measured in the test case or additional test cases are required. [1] measured code coverage in terms of statement and branch directly helping to support our study in measuring whether coverage is strongly correlated with test suite effectiveness.

Another metric being calculated is defect density which is measured by number of

defects upon the total lines of code. Lines of code varies version wise, but measure helps in identifying defects earlier rather than late which helps in earlier maintenance rather investing it later.

Based on the finding our motivation here is to measure data for the selected metrics and analyze using correlation analysis whether the above metrics relate with each other and if yes what affect they cause and how it matches with our findings. So, to prove what we are trying to measure, metrics were formed with a general hypothesis. At first, we are trying to validate if

i) Test Suite with higher coverage might show better test suite effectiveness.
ii) Classes with higher complexity are less likely to have higher coverage test suites.
iii) Class with low coverage contains more defects.
iv) Higher the maintainability the lower is defect density.

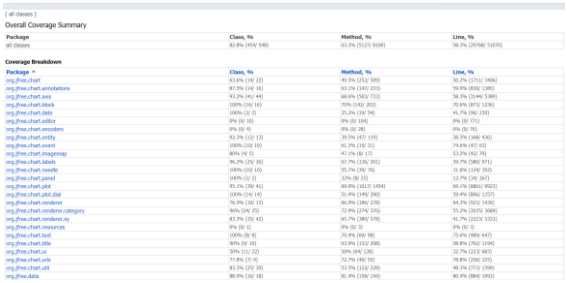
Based on the above formed hypothesis, plugins and tools were used to measure required metrics and data analyzed.

3.METHOD

3.1 Data Collection

Data collection was done is multiple steps. First step was to search for open source projects that contains more than 100K LOC as requirements stated. From this condition, open source java projects were imported from GitHub. Secondly every project gave errors while importing projects through git url. Some of the dependencies were not resolved by directly importing project through git URL so every project had to be installed via zipping and extracting it in the local. First step of project filtration was done. Then projects were imported by the pom.xml file in (Intellij IDEA) as all the projects selected were maven build. This resulted in resolving dependencies as JDK version 8 and later supported all dependencies after project build. To calculated code coverage, Intellij provides an inbuilt code coverage tool named Clover which calculated line, statement, branch coverage in one go. So as mentioned above, each project's 5 versions were selected and extracted to calculated code coverage. Before running the tool, maven projects had to be clean build. Maven project were built using maven clean install from the Intellij command prompt. This resolves any additional dependencies as well. Then code coverage tool was selected by right clicking on test cases and then selecting option of Run all test with coverage. The coverage option can be adjusted through edit configuration under Run option in tool bar. After further research, Intellij also supports JaCoCo code coverage tool. For this project, former code coverage tool was used. Clover was not detecting code coverage in some of our project so some modification to pom file was necessary like addition of clover dependency in pom file sorted out the the error. Results of code coverage outputs every test class's coverage for every condition/statement

within the whole project. Below is the screenshot of the result from code coverage


[all classes]

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	83.8% (454/ 540)	61.7% (5127/ 8308)	58.3% (29766/ 51070)

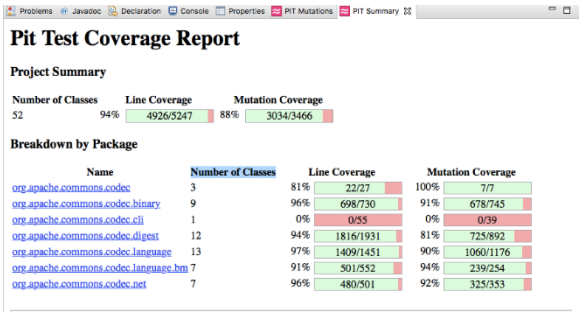
Coverage Breakdown

Package ^	Class, %	Method, %	Line, %
org.jfree.chart	63.6% (14/ 22)	49.5% (252/ 509)	50.2% (1711/ 3406)
org.jfree.chart.annotations	87.5% (14/ 16)	61.1% (147/ 240)	59.8% (830/ 1385)
org.jfree.chart.axis	93.2% (41/ 44)	68.6% (503/ 733)	58.3% (2144/ 3680)
org.jfree.chart.block	100% (16/ 16)	70% (142/ 203)	70.6% (872/ 1236)
org.jfree.chart.date	100% (2/ 2)	35.2% (19/ 54)	41.7% (96/ 230)
org.jfree.chart.editor	0% (0/ 10)	0% (0/ 164)	0% (0/ 771)
org.jfree.chart.encoders	0% (0/ 4)	0% (0/ 28)	0% (0/ 76)
org.jfree.chart.entity	92.3% (12/ 13)	39.5% (47/ 119)	38.5% (168/ 436)
org.jfree.chart.event	100% (16/ 16)	61.3% (19/ 31)	74.6% (47/ 63)
org.jfree.chart.imagemap	80% (4/ 5)	47.1% (8/ 17)	53.2% (42/ 79)
org.jfree.chart.labels	96.2% (25/ 26)	67.7% (136/ 201)	59.7% (380/ 637)
org.jfree.chart.needle	100% (10/ 10)	55.7% (39/ 70)	51.6% (124/ 240)
org.jfree.chart.panel	100% (2/ 2)	10% (3/ 31)	12.7% (34/ 267)
org.jfree.chart.plot	95.1% (39/ 41)	60.9% (1017/ 1454)	69.1% (3881/ 5615)
org.jfree.chart.plot.dial	100% (14/ 14)	51.4% (149/ 290)	59.4% (886/ 1357)
org.jfree.chart.renderer	76.9% (10/ 13)	66.9% (186/ 278)	64.3% (921/ 1430)
org.jfree.chart.renderer.category	90% (18/ 20)	72.9% (274/ 376)	55.2% (2870/ 5089)
org.jfree.chart.renderer.xy	83.3% (15/ 42)	65.7% (380/ 578)	41.7% (2222/ 5333)
org.jfree.chart.resources	0% (0/ 1)	0% (0/ 2)	0% (0/ 3)
org.jfree.chart.text	100% (8/ 8)	70.4% (69/ 98)	75.6% (489/ 647)
org.jfree.chart.title	90% (9/ 10)	63.9% (132/ 206)	58.8% (762/ 1296)
org.jfree.chart.ui	50% (11/ 22)	50% (64/ 128)	32.7% (223/ 683)
org.jfree.chart.urls	77.8% (7/ 9)	72.7% (40/ 55)	78.8% (256/ 325)
org.jfree.chart.util	81.3% (26/ 32)	53.5% (122/ 228)	48.1% (772/ 1604)
org.jfree.data	88.9% (16/ 18)	81.4% (158/ 194)	80.9% (884/ 1093)

Fig 1: Code coverage result

Required data was extracted from each class for relating if test cases covered all possible conditions or not. Some of the projects were eliminated due to not having test cases or even present having faced difficulty in resolving dependency errors.

For mutation score, Eclipse IDE was used as plugin supported in Intellij did not print the expected result (discussed in results section) so plugin was changed. Using Pitclipse, mutation score was calculated for every mutant being covered. Mutations score is calculated by measuring calculating killed mutants upon total number of mutants * 100. Pitclipse calculates the overall mutations score and for every individual class as well. An example of mutation score is shown below.



Problems Javadoc Declaration Console Properties PIT Mutations PIT Summary

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
52	94%	4926/5247	88%	3034/3466

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.apache.commons.codec	3	81%	22/27	100%	7/7
org.apache.commons.codec.binary	9	96%	698/730	91%	678/745
org.apache.commons.codec.cli	1	0%	0/55	0%	0/39
org.apache.commons.codec.digest	12	94%	1816/1931	81%	725/892
org.apache.commons.codec.language	13	97%	1409/1451	90%	1060/1176
org.apache.commons.codec.language.bm	7	91%	501/552	94%	239/254
org.apache.commons.codec.net	7	96%	480/501	92%	325/353

Fig 2: Mutation testing result

Correlation between mutation score and code coverage needs to be found out.

Cyclomatic complexity is the measure of complexness of the code. Cyclomatic complexity measures the number of independent linearly independent paths through program source code. For data collection cyclomatic complexity was measured by metrics-reloaded plugin easily installable in Intellij IDE. This measure average and total cyclomatic complexity for individual project at class level, method level. For analysis part cyclomatic complexity was used at class level. Reason for taking the data at class level was to analyze if complexity of class is higher than does the test suite covers that complexity making it more efficient or less efficient. To install the plugin, in the Intellij IDE under files tab, there is an option of plugin. In the plugins tab a window appears where it shows the listed plugins available at marketplace and installed. Under marketplace tab, in search tab MetricReloaded was searched and installed. After installation IDE restarts. To calculate the metric, what was done is at home of the IDE, under help tab the first option i.e. Find Action was selected, and a floating search bar appears where Calculate metrics needs to be typed. It gives an option to calculate whole project metric or file wise. And below that option is a drop-down titled Complexity Metric besides which desired metrics can be selected and after clicking OK the result is displayed on the console.

Maintainability index is nothing but measure of maintenance of software at class level or method level. This was calculated using formula: $171-5.2 * \ln(V) – 0.23 * (G) – 16.2 * \ln(LOC)$. Here V means Halsted Volume, G means Cyclomatic complexity and LOC means lines of code. Cyclomatic complexity and lines of code were calculated from MetricReloaded for previous analysis as a metric and Halstead volume was also calculated from MetricsReloaded. Halstead has similar type of complexity measures but in also calculates in terms of volume, effort and difficulty. Halstead volume is calculated by program length which is sum of number of operators and operands in the program multiplied by logarithmic base to 2 of program vocabulary i.e. sum of distinct operators and operands. The calculation wasn't needed to be done manually as the plugin supported so class wise data was collected.

Defect density is the measure of total number of defects divided by total number of lines of code. For defect density some popular tools such as GitHub Issue tracker and JIRA [4] were used to find problems reported and sort out bug from the report. To collect defects from GitHub issue tracker, Issues tab is listed in some of the projects so here it was made available for JFreeChart. For data purpose, labels such as bugs, defects count was collected. This count is displayed on top of the Issues page.

Following table describes the collected data for each metric.

Projects	Code Coverage	Complexity	Maintainability Index	Defect Density
Jfreechart_1.5	61	2.06	38.79731	4.80708
Jfreechart_1.18	58.3	1.94	36.5764	4.51528
Jfreechart_1.19	54.8	1.99	37.8123	3.36233
Common_Codec_1	94.8	1.89	65.42703334	2.65906
Common_Codec_2	95	2.04	69.43601814	3.22886
Common_Codec_3	92.9	1.99	68.89013312	3.22886
Common_Codec_4	94.1	1.92	65.77762332	3.22886
Common_Codec_5	94.2	1.89	65.62086983	3.22886
Common_io_1	89.2	1.84	65.3112813	8.38806
Common_io_2	89.2	1.83	69.34929972	8.71318
Common_io_3	89.2	1.84	68.37315075	8.71318
Common_io_4	88.4	1.85	74.51427926	8.71318
Common_io_5	89.5	1.83	65.84537563	7.99823

Table 1: Data collected project wise

3.2 Analysis

For our analysis of hypothesis, it was necessary to generalize the data and perform some of correlation to understand the effects of each metric on one another. The data being irrational, there had to be a process to rationalize the data and perform analysis. Much effort and work would have gone into rationalizing the data which had made loose our focus from the main aim of project i.e. to understand effects caused on by some of the metrics on each other. So, Spearman rank order correlation was selected as it is one of the Pearson's product moment correlation also is better for measuring strength and direction of association between two ranked variables.

Ranking helps in finding the correlation between ordinal, interval or ratio scale. Here our data was on an interval scale, so it made our task easier. One principle of spearman's correlation is if there are no data values being repeated then a perfect spearman correlation of +1 or -1 occurs when each variable is perfectly monotone function of other. The formula for calculating spearman correlation is

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

Where n is the number of observations and d_i is the difference between two ranks of each observations. Spearman's

interpretation works as if Y tends to increase when X increases the spearman correlation is positive. If Y tends to decrease when X increases, then correlation is negative. A 0 value indicate there is no tendency for Y to either increase or decrease when X increases.

4.RESULT AND DISCUSSION

Based on the above analysis method, correlation was found out in 4 different parts.

1)Test Suite with higher coverage might show better test suite effectiveness.

Test suite coverage for individual projects was calculated. Coverage calculates package wise summing total statement and branch coverage. Within package class wise coverage is calculated. Below graph shows

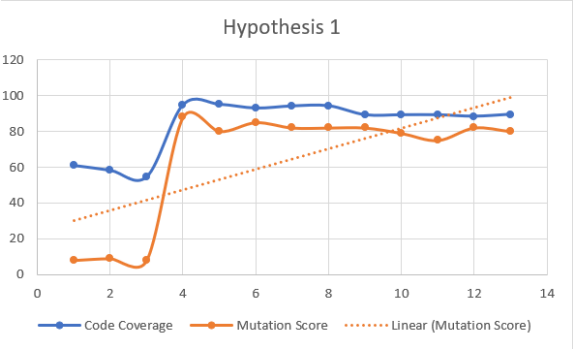


Fig 3: Correlation for hypothesis 1

Y(Code coverage) increases when X(Mutation score) increases, there is a positive correlation between both. As per spearman's calculation value of the relation is 0.37. The value is neither 1 nor -1 but falls between positive relation predicting that test suite with higher coverage show better test suite effectiveness.

2) Classes with higher complexity are less likely to have higher coverage test suites.

For complexity, Halstead's cyclomatic complexity is measured in comparison to coverage of classes. Overall complexity for all projects was calculated on average at class level. This complexity supports the one Halstead proposed for identifying complexness of classes. The correlation between complexity and code coverage predicts the following.

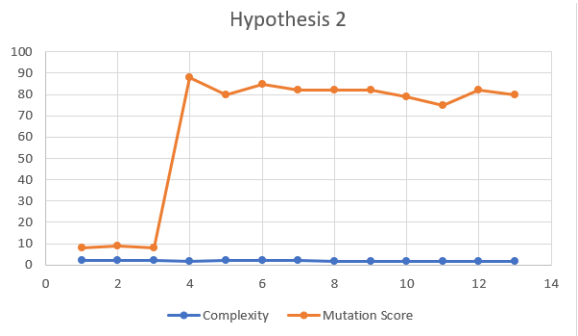


Fig 4: Correlation of Hypothesis 2

The complexity of the projects is linear, but mutation score is more than the complexity, so the graph represents like this. Spearman correlation for these two metrics is 0.81 which is close to 1 predicting positive relation between these two as higher the complexity, the less likely to have higher test suite effectiveness.

3) Class with low coverage contains more defects.

Coverage at class level was considered from each different module as to prove this hypothesis. Defect density was calculated using JIRA and GitHub issue tracker. The correlation to be found out here is lower the coverage of class in terms of statement and branch the more the defects that class contains. Spearman's rank correlation for above two metrics was 0.16 which is a little weaker that the other two hypotheses being found out. Below graph depicts comparison

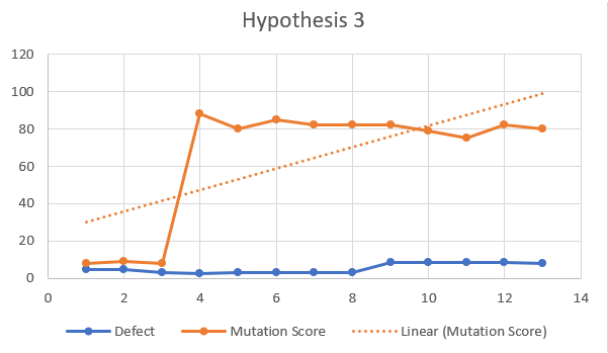


Fig 5: Correlation of Hypothesis 3

At first defect density is equal to mutation score but afterwards mutation score increases and defect density lowering predicts classes with low coverage contain equal defects which somewhat matches with the expected findings.

4) Higher the maintainability the lower is defect density.

Maintainability index was calculated using [7] standard formula. The rationale for this hypothesis was to measure maintainability index and compare it with defect density to check whether defect density is lower when projects maintainability index is higher. To our analysis maintainability index was higher for some of the versions but defect density remained somewhat same for different versions. Spearman correlation for above metric calculation was found out to be 0.18 which also predicts a positive but not a stronger relation from the below graph.

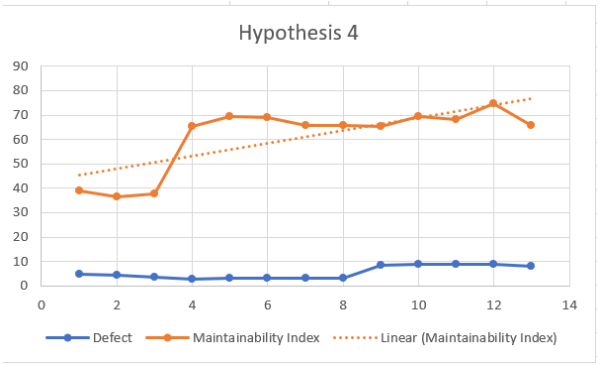


Fig 6: Correlation of Hypothesis 4

Above correlation observation is done as the number of maintainability index increases, defect density reduces. From the spearman correlation, analysis does satisfy but with a weaker correlation of increase in defect density.

5. CHALLENGES

While doing analysis part, it was unclear whether the analysis was done in a proper manner or some factors were missing. So, from start, during project selection our work of selecting 3 different open source java projects increased our work as the first batch selected did not have test cases. Second batch of projects consisted of test cases but were not identified by the IDE during running code coverage tool. So, every time none of the metrics were found, project list was changed but for JFreechart only 3 versions were found so some of the calculations might vary if project size increases. This increased our time by 2x. Metric for 5 and 6 was to be found out but to do so, some of the literature work had to be studied. Without knowing any metric what can be done. A thorough research had to be done. Maintainability index was a known metric for calculating at project level, but multiple formulae made it difficult to calculate it. So, a genuine formula was chosen to work on.

The data collected is for 5 versions of 3 different project which might not be enough to support analysis but within these collections some analysis can be done to verify if supported data proves correlation between defined metrics.

6. CONCLUSION

From the findings it became clear that some of the data did support correlation of software quality attributes. From coverage point of view, it was better understood that classes with low coverage can contain less defects but from maintainability point of view classes with lower defects are somewhat maintainable as for one project maintainability is greater than 80 which is considered highly maintainable but for other two measures it is considered less maintainable.

From the collected metrics and hypothesis, software quality attributes do relate with each other from different perspective and vary accordingly which can help in improving overall quality and maintenance effort required for that software reducing time and cost.

7.Acknowledgement

We would like to thank our professor Dr Jinqiu Yang for allowing us to take this project and helping us with our doubts and project selection.

8.References

[1] Laura Inozemtseva and Reid Holmes. 2014," Coverage is not strongly correlated with test suite effectiveness". In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, New York, NY, USA

[2]M H Moghadam, S M Babamir "Mutation score evaluation in terms of object-oriented metrics", 4th International conference on computer and knowledge engineering (ICCKE), 2014, Iran

[3] H M. Olague, L H. Etzkorn, G Cox,"An entropy based approach to assessing object oriented software maintainability and degradation- A method and case study", University of Alabama, USA

[4] JIRA Issue Tracker, Available [Online] https://issues.apache.org/jira/projects/CODEC/issues/CODEC-134?filter=allopenissues

[5] GitHub: Apache Commons IO. Available [Online]https://github.com/apache/commons-io

[6]GitHub: Apache Commons Codec. Available[Online]https://github.com/apache/commons-codec

[7] Maintainability Index, ProjectCodeMeter Available[Online]:http://www.projectcodemeter.com/cost_estimation/help/GL_maintainability.htm