SOEN 6611

SOFTWARE MEASUREMENT

WINTER 2020

Professor : Jinqiu Yang

Project Report

By

Team-N

GITHUB : https://github.com/RVPKR777/SOEN-6611_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

Swetha Chenna
Department of Sotware Engineering
Concordia University
Montreal,Canada
swethachenna2018@gmail.com

Venkata Pavan Kumar Reddy Ravi
Department of Sotware Engineering
Concordia University
Montreal,Canada
pavan.03121996@gmail.com

Nandini Bandlamudi
Department of Sotware Engineering
Concordia University
Montreal,Canada
nadu.angel555@gmail.com

Gurminder Pal Dhiman
Department of Sotware Engineering
Concordia University
Montreal,Canada
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.**

I. 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.

II. OPEN SOURCE PROJECTS SELECTED

We selected three open source projects one is below 100k lines of code and two are above 100k lines of code according to requirements specified. And the explanation about all these three projects will be specified below:

A. Apache Commons Configurations*:*

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.

Size:122K

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

Technology: Java

B. Apache Commons Collections*:*

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.

Size:118K

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

Technology: Java

C. Apache Commons DbUtils*:*

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.

Size:15K

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

Technology: Java

III. METRICS

Below are the metrics selected to measure the software projects selected above.

A. **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

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

2) Data Collection and Analysis:

a) Tools Used: JaCoCo

b) Reason why we used JaCoCo: Because it is a free code coverage tool available in the internet which satisfies statement coverage ,Branch coverage and complexity requirements.

c) How to do: 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.

Apache Commons Configuration

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.configuration2.plist		64%		54%	317	635	519	1,479	56	198	1	18
org.apache.commons.configuration2		90%		89%	261	1,646	296	3,619	137	1,008	0	74
org.apache.commons.configuration2.io		78%		74%	72	389	158	879	16	228	1	30
org.apache.commons.configuration2.beanutils		86%		84%	34	233	50	498	3	116	0	9
org.apache.commons.configuration2.resolver		70%		54%	19	53	37	152	1	29	0	4
org.apache.commons.configuration2.interpol		89%		88%	15	117	26	258	9	82	0	13
org.apache.commons.configuration2.convert		95%		90%	28	212	17	420	2	72	0	9
org.apache.commons.configuration2.tree		98%		95%	35	740	14	1,575	5	418	0	48
org.apache.commons.configuration2.builder.combined		97%		93%	19	310	14	771	4	200	0	20
org.apache.commons.configuration2.web		79%		72%	7	34	10	59	3	23	1	6
org.apache.commons.configuration2.tree.xpath		96%		94%	9	151	16	305	3	83	0	9
org.apache.commons.configuration2.reloading		95%		100%	0	85	6	177	0	54	0	9
org.apache.commons.configuration2.event		98%		97%	3	106	4	224	1	68	0	9
org.apache.commons.configuration2.builder		99%		98%	4	303	4	638	1	200	0	24
org.apache.commons.configuration2.ex		85%		n/a	2	12	4	24	2	12	0	3
org.apache.commons.configuration2.builder.fluent		100%		100%	0	61	0	77	0	56	0	3
org.apache.commons.configuration2.sync		100%		100%	0	14	0	23	0	13	0	3
Total	5,705 of 45,968	87%	744 of 4,416	83%	825	5,101	1,175	11,178	243	2,860	3	291

Apache Commons Collections

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.collections4.map		89%		79%	357	1,808	276	3,172	71	887	2	103
org.apache.commons.collections4.trie		76%		71%	156	502	189	918	33	210	1	24
org.apache.commons.collections4.bidimap		89%		87%	85	573	105	1,170	38	335	0	35
org.apache.commons.collections4.iterators		87%		89%	104	585	149	1,115	67	385	3	46
org.apache.commons.collections4		92%		88%	137	1,008	114	1,535	63	621	0	55
org.apache.commons.collections4.multiset		74%		59%	83	217	103	489	28	138	0	17
org.apache.commons.collections4.functors		85%		85%	83	398	95	685	45	240	0	55
org.apache.commons.collections4.list		94%		89%	80	578	65	1,224	15	342	0	26
org.apache.commons.collections4.set		88%		87%	40	200	49	456	28	209	0	17
org.apache.commons.collections4.multimap		90%		78%	37	282	48	485	13	205	0	24
org.apache.commons.collections4.comparators		85%		69%	55	151	34	229	11	84	0	8
org.apache.commons.collections4.bag		91%		88%	39	254	39	474	20	165	0	18
org.apache.commons.collections4.collection		93%		92%	21	198	28	370	11	135	0	9
org.apache.commons.collections4.sequence		90%		82%	10	73	19	141	5	39	1	8
org.apache.commons.collections4.splitmap		83%		66%	7	38	9	47	5	27	0	2
org.apache.commons.collections4.queue		97%		92%	6	112	8	203	6	70	0	7
org.apache.commons.collections4.bloomfilter		96%		94%	6	130	1	283	0	80	0	8
org.apache.commons.collections4.keyvalue		98%		84%	15	125	5	189	1	81	0	8
org.apache.commons.collections4.trie.analyzer		82%		70%	11	25	4	40	1	8	0	1
org.apache.commons.collections4.properties		98%		92%	3	70	4	122	2	63	0	6
org.apache.commons.collections4.bloomfilter.hasher.function		96%		100%	0	33	3	62	0	29	0	5
org.apache.commons.collections4.bloomfilter.hasher		100%		100%	0	84	0	188	0	58	0	12
Total	6,009 of 56,777	89%	1,035 of 5,975	82%	1,297	7,377	1,386	13,514	457	4,381	7	482

Apache Commons DbUtils

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.dbutils		58%		77%	280	501	521	1,161	235	435	7	30
org.apache.commons.dbutils.wrappers		100%		93%	2	66	0	128	0	53	0	2
org.apache.commons.dbutils.handlers		100%		100%	0	58	2	114	0	48	0	12
org.apache.commons.dbutils.handlers.columns		100%		87%	5	44	0	30	0	33	0	10
org.apache.commons.dbutils.handlers.properties		100%		93%	1	16	0	24	0	5	0	2
Total	2,048 of 5,580	66%	87 of 388	77%	308	704	521	1,487	235	573	7	64

B. **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

1) Formula: Number of Decision Statements executed/Total number of decision outcomes*100

2) Data Collection and Analysis:

a) Tools Used: JaCoCo

b) Reason why we used JaCoCo: Because it is a free code coverage tool available in the internet which satisfies statement coverage ,Branch coverage and complexity requirements.

c) How to do: 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.

C. **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.

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

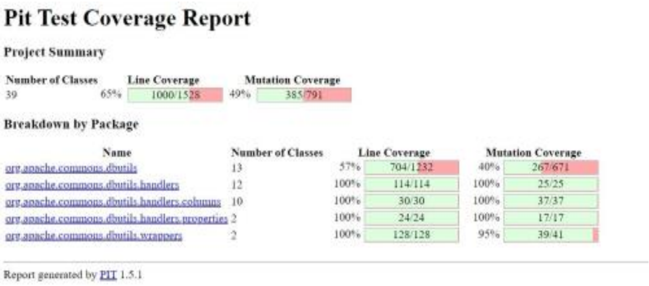
2) Data Collection and Analysis:

a) Tools Used: Maven-Pitest plugin

b) Reason why we used Pitest Plugin: Easy to get the mutation using plugin when compared other procedures

c) How to do: 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.
- Mutation Coverage 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.



Pit Test Coverage Report

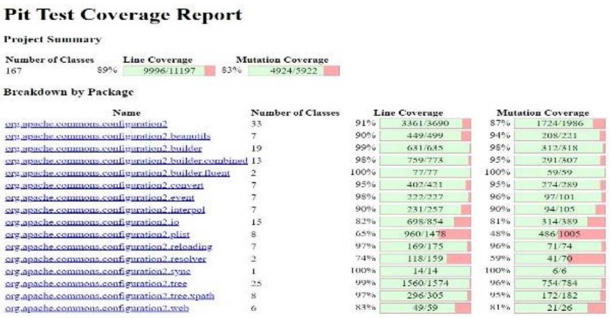
Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
39	65%	1000/1528	49%	385/791

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.apache.commons.dbutils	13	57%	704/1232	40%	267/671
org.apache.commons.dbutils.handlers	12	100%	114/114	100%	25/25
org.apache.commons.dbutils.handlers.columns	10	100%	30/30	100%	37/37
org.apache.commons.dbutils.handlers.properties	2	100%	24/24	100%	17/17
org.apache.commons.dbutils.wrappers	2	100%	128/128	95%	39/41

Report generated by PIT 1.5.1



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
167	89%	9996/11197	83%	4924/5922

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.apache.commons.configuration2	33	91%	3361/3690	87%	1724/1986
org.apache.commons.configuration2.beanutils	7	90%	449/499	94%	208/221
org.apache.commons.configuration2.builder	19	99%	631/635	98%	312/318
org.apache.commons.configuration2.builder.combined	13	98%	759/773	95%	291/307
org.apache.commons.configuration2.builder.fluent	2	100%	77/77	100%	59/59
org.apache.commons.configuration2.convert	7	95%	402/421	95%	274/289
org.apache.commons.configuration2.event	7	98%	222/227	96%	97/101
org.apache.commons.configuration2.interpol	7	90%	231/257	90%	94/105
org.apache.commons.configuration2.io	15	82%	698/854	81%	314/389
org.apache.commons.configuration2.plist	8	65%	960/1478	48%	486/1005
org.apache.commons.configuration2.reloading	7	97%	169/175	96%	71/74
org.apache.commons.configuration2.resolver	2	74%	118/159	59%	41/70
org.apache.commons.configuration2.sync	1	100%	14/14	100%	6/6
org.apache.commons.configuration2.tree	25	99%	1560/1574	96%	754/784
org.apache.commons.configuration2.tree.xpath	8	97%	296/305	95%	172/182
org.apache.commons.configuration2.web	6	83%	49/59	81%	21/26

D. 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 actions

1) Formula: E-N+2

Where E= Number of edges in graph

N=Number of nodes in graph

CC=Cyclomatic Complexity

2) Data Collection and Analysis:

a) Tools Used: JaCoCo

b) Reason why we used JaCoCo: Because it is a free code coverage tool available in the internet which satisfies statement coverage ,Branch coverage and complexity requirements.

c) How to do: 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.

E. 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.

1) Formula: 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

2) Data Collection and Analysis:

a) Tools Used: Jhawk

b) Reason why we used Jhawk: Because it gives maintainability index of method, class and package which is good to analyse in detail.

c) How to do: 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

F. **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.

1) Formula: Number of defects/Size of release SLOC

2) Data Collection and Analysis:

a) Tools Used: JIRA and LocMetric

b) Reason why we used JIRA and LocMetric: Retrieving data from official site to get accurate count of bugs and LocMetric which is easy to adapt.

c) How to do: 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. Click analyze and run the project in the tool.

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

IV. 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 (d^2) 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.

Project Name	M1-Statement Coverage	M2-Branch Coverage	M4-Average Cyclomatic Complexity	Spearman Correlation (M1 & M4)	Spearman Correlation (M2 & M4)
Apache Commons Collections	51%	82%	12.3	0.42417	-0.16242
Apache Commons Configuration	89%	83%	14.7	0.04871	-0.23174
Apache Commons DbUtils	64%	77%	7.1	-0.66689	-0.5

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.

Project	Code Coverage	Mutation Score	Spearman Correlation Coefficient
Apache Commons Collections	51%	43%	0.82391
Apache Commons Configuration	89%	80%	0.9142
Apache Commons DbUtils	64%	47%	0.3421

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.

Project Name	Statement Coverage	Branch Coverage	Number of Bugs
Apache Commons Collections	51%	82%	5
Apache Commons Configuration	89%	83%	11
Apache Commons DbUtils	64%	77%	6

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.

Projects	Versions	Statement Coverage	Branch Coverage	Number of bugs	Spearman Coefficient (M1 & M6)	Spearman Coefficient (M2 & M6)
Apache Commons DbUtils	1.1	56%	51%	4	r_s = -0.22361	r_s = 0.22361
	1.2	63%	53%	4		
	1.4	79%	66%	2		
	1.6	57%	64%	4		
	1.7	64%	77%	6		
Apache Commons Configuration	1.8	77%	70%	16	r_s = -0.5	r_s = -0.5
	2.0	68%	63%	11		
	2.2	86%	56%	7		
	2.4	79%	77%	2		
	2.6	85%	81%	4		
Apache Commons Collections	2.0	65%	49%	4	r_s = 0.8	r_s = 0.4
	3.0	76%	68%	15		
	3.2	86%	81%	55		
	4.1	69%	77%	16		
	4.4	51%	82%	5		

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

Correlation between Maintainability Index and Post release defect density:

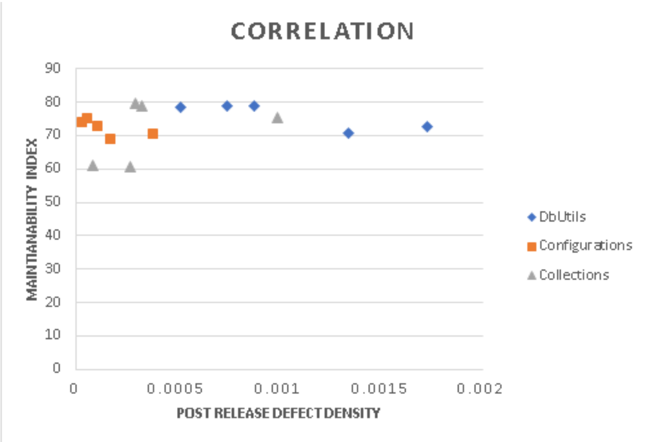
We have started the correlation with a hypothesis that with a high maintainability index we will have low post release defect density value**.**

Projects	Versions	Maintainability Index	Defect density	Spearman coefficient
Apache Commons DbUtils	1.1	72.63	0.00173	r_s = -0.5
	1.2	70.57	0.00134	
	1.4	78.43	0.000514	
	1.6	78.79	0.00074	
	1.7	79.03	0.000874	
Apache Commons Configuration	1.8	70.41	0.000379	r_s = -0.8
	2.0	68.92	0.000169	
	2.2	72.6	0.000105	
	2.4	73.92	0.0000297	
	2.6	75.16	0.0000592	
Apache Commons Collections	2.0	79.62	0.000293	r_s = 0.5
	3.0	78.96	0.000326	
	3.2	75.24	0.000992	
	4.1	60.56	0.000263	
	4.4	61.06	0.0000789	

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 for Post 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:

- G. K.Gill and C. F. Kemerer, "Cyclomatic complexity density and software maintenance productivity," IEEE Trans. Software Eng, vol. 17, no. 12, Dec 1991
- 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.
- L. Vinke, "Estimate the post-release Defect Density based on the Test Level Quality," 2011.
- Relating Code Coverage, Mutation Score and Test Suite Reducibility to Defect Density https://ieeexplore.ieee.org/document/7528960 [7] Sharma, R. (2014).
- Variations of Maintainability Values formula http://www.virtualmachinery.com/sidebar4.htm
- 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.
- Spearman's coefficient:

  https://geographyfieldwork.com/SpearmansRank.htm
- 2006 30th Annual International Computer Software and Applications Conference (COMPSAC '06 Supplement)
- M.B. O'Neal, W.R. Edwards, "Complexity measures for rule-based programs", Knowledge and Data Engineering IEEE Transactions on, vol. 6, no. 5, pp. 669-680, 1994