Analysis and Correlation of Software metrics

Software Measurement SOEN-6611 Winter 2020, Team S

Subhannita Sarcar
Department of Computer Science and Software Engineering
Concordia University, MEng
Montreal, Canada
subhannitasarcar1994@gmail.com

Gurwinder Kaur
Department of Computer Science and Software Engineering)
Concordia University, MEng
Montreal, Canada
kaurgurwinder2013@gmail.com

Jasmine Kaur
Department of Computer Science and Software Engineering
Concordia University
Montreal, Canada
jasmine14concordia@gmail.com

Raghav Sharma
Department of Computer Science and Software Engineering
Concordia University, MEng
Montreal, Canada
raghav.1995@yahoo.com

***Abstract*—Software testing has long been established as an integral component of the software development lifecycle. The progress and quality of the testing effort can be measured and quantified with the help of testing metrics. The objective of this project is to measure 6 such metrics from 3 open-source projects and correlate the results. The correlation would give an indication of the trends of testing efforts and the possibility of predicting one metric through another.**

Keywords—software testing metrics, test coverage, open-source project, metric correlation

I. Introduction

Software characteristics are diverse and require precise classification to streamline its measurement. Considerable research in this field has been able to identify the measurable quantities and how they can be used to identify the quality of software in empirical terms. In this project, we focus on the metrics used to evaluate testing efforts. These metrics prove beneficial in assessing developer performance or productivity, the accuracy of test suite, future costs and work items, support of managerial decisions that further aid in risk assessment. Since the fundamental reason behind tracking the quality of a process is to gain improvements for higher efficiency, software managers are using metrics to gain an increased return on their investment, reduce overtime and costs among other factors [1].

Although software metrics and its related activities have seen significant research with exponential growth in published works and almost as many books as there are on the general topic of software engineering, the degree of overlap between academic and industrial practices remain low [2]. This is mainly due to either irrelevance of the metric in terms of scope or content or poorly motivated and executed metrics activity. It is evident that an industrial practice will not be adopted unless a clear correlation between the software metric and business value is exploited. A study [3] has proposed a validation framework from the perspective of measurement method that takes into account the details of a measurement's process model to verify the relationship between the definitions of the concept and the numerical assignment rules as well as the reproducibility of the metric.

In this research, we have investigated three open-source projects and measured 6 software metrics against them. We have attempted to establish consistency between the metrics. This is because it has been established that internal software metrics, when used as the sole predictors, are poor at coming to reliable conclusions about software quality [4]. Most of the related work is focussed on predicting faults and failures in the program. A parallel study [5] has drawn comparisons between the usage of design metrics and code metrics for fault predictions through statistical relationships and expert knowledge. This is based on the hypothesis that design (such as McCabe complexity) and code metrics (such as lines of code) are equally good from a prediction point of view, but design metrics will be able to make the prediction sooner, thereby incurring a lower cost.

The open-source projects selected are as follows –

- Apache Commons Collections (P1):

  Source code repository: https://gitbox.apache.org/repos/asf?p=commons-collections.git. Issue tracking system: http://issues.apache.org/jira/browse/COLLECTIONS This framework is responsible for the inclusion of powerful data structures, interface and implementation utilities to improve the development of Java applications and has become the established benchmark for collection handling in Java [6]. The versions used for this project are – 4.0, 4.1, 4.2, 4.3, 4.4.

- Apache Commons FileUpload (P2):

  Source code repository: https://gitbox.apache.org/repos/asf?p=commons-fileupload.git. Issue tracking system: https://issues.apache.org/jira/browse/FILEUPLOAD

  This package has added utilities for efficient file upload services to servlets and web applications. The breakthrough introduced through FileUpload package is parsing of POST method from a form that has a content type of multipart/form-data so that the results remain flexible to the caller's features [7]. The versions used for this project are – 1.2, 1.2.2, 1.3, 1.4.

- Apache Commons IO (P3):

  Source code repository: https://gitbox.apache.org/repos/asf/commons-io.git. Issue tracking system:

https://issues.apache.org/jira/browse/IO. This library consists of utility classes that develop IO functionality [8]. The versions used for this project are – 2.2, 2.3, 2.4, 2.5, 2.7.

All the above selected open-source projects are written in Java with JUnit 3 or 4 test cases and are built with Maven automation tool. The plugin version of Junit is maintained at least 4.6 since the tool used for mutation score calculation (PIT) can only work a minimum version of JUnit 4.6 and JUnit 4 is configured to run JUnit 3 test cases as well.

II. Selection Of Metrics And Data Collection

The following section describes the metrics selected for this project, the rationale behind their selection, data collection steps and related work:

A. Statement coverage

A test case is considered to be the input based on which the program under test is executed and validated. A test set is a set of such test cases. The notion of the adequacy of statement coverage depends on the percentage of executable statements that are covered by tests. A test set that satisfies the requirement of executing all the statements at least once is said to attain 100% coverage under the statement coverage criterion [9].

The statement coverage criterion is said to be satisfied by a set P of execution paths if at least one path p in P contains the node n, for all nodes n in the control flow graph [9]. The Java Code Coverage library or JaCoCo project hosted within EclEmma [10] has been used to calculate the statement coverage for all classes in our open source projects. Jacoco derives coverage counters from information contained in Java classes and uses this information to calculate coverage metrics. Statement coverage (SC) is calculated as:

$$Statement\ coverage\ SB = \frac{Lines\ covered}{Lines\ covered + Lines\ missed} \quad (1)$$

Table I (column SC) shows that statement coverage steadily increases with every version in the three projects. This could result from better code practices or the usage of modern tools that show source code lines which are uncovered.

It has been proved that a test case that exposes new program elements, or rather takes the effort of exploring a maximum number of statements is likely to be better at revealing possible faults [11]. This relationship holds, irrespective of the test suite size. It is more evident in large size projects that higher test coverage leads to an increase in defect coverage [12]. This knowledge of reliability growth can help developers to allocate the right amount of resources in the right direction to achieve the target reliability [13].

B. Branch Coverage

Statement coverage relies on the proposition that a testing effort is not considered sufficiently good if every reachable code block is not executed, at least once, by the test suite. This practice is regarded as the least adequacy criterion. It is often observed that better testing was attainable without going through the painstaking efforts of making sure that every statement is covered. Alternatively, there are usually different code elements such as predicates (especially in complex programs or large-size industrial software), where faults may remain unexposed if a certain condition is not fulfilled. These alternative branches of code execution can be explored using Branch coverage [14].

Branch coverage intends to find a set of test cases that exercises execution over a maximum number of branches in the software under test. One study has presented the problems in such a single-objective target and has shown that when each branch is viewed as an objective to be optimized, the reformulated branch coverage is able to attain higher coverage or faster convergence [15]. In this project, we have simply calculated Branch coverage (BC) as [16]:

$$Branch\ coverage\ BC = \frac{Branches\ covered}{Branches\ covered + Branches\ missed} \quad (2)$$

Jacoco reports are used to get information on the branches covered and missed counters. It must be noted that *if* and *switch* statements were treated as predicate nodes and exception handling was ruled out.

Table I (column BC) shows that branch coverage mostly remains steady with each version within a certain tolerance. It was found that several classes had 0 branches, i.e., a single branch of execution. These classes were not taken into account while calculating branch coverage on the class level. However, version level branch coverage was directly taken from the Jacoco report's overall branch coverage percentage counter.

The utility of test coverage lies in its correlation with software reliability and the strength of test cases in detecting a greater number of defects. Although a higher coverage value can be directly correlated to a more reliable code, it is also observed in large scale industrial projects that increasing test coverage does not provide additional benefits after a certain point because there exist some program blocks that are more frequently exercised than the others [17].

C. Mutation score of test suite

Code coverage practices are important indicators of the thoroughness of a test suite, i.e., the detail up to which the test cases comb through the program and follows the ideology of regression testing. However, this does not indicate the effectiveness or accuracy of the test suite.

Mutation testing is a fault-based testing technique which aims to expose errors frequently made by developers. This is done by introducing or seeding faults into the program through syntactic analysis to deliberately create faulty programs or mutants. The test cases are run against these faulty programs, and it is verified if the results of the test cases are different for the original programs and mutants, in which cases the mutants are identified [18]. If the mutant survives, i.e., the test cases fail to identify the changed code, it is either because the mutations did not result in any change in functionality or the mutants are not equivalent. In the former case, it is impossible for any test case to kill the mutant, whereas in the latter case, the test case truly failed [19].

Mutation score (MS) is calculated as the ratio of killed or detected mutants to the total number of non-equivalent and non-covered mutants [20]. We have used the PIT testing tool [21] to find the mutation score of all classes in our open source projects. This tool manipulates bytecode to generate mutants by scanning the main controlling process and recording the location and type of mutants. Furthermore, it runs only the tests that may kill the mutants, thereby achieving speed [22].

Table I (column MS) shows that the mutation score either remains steady or improves with each version, suggesting

that test cases were improved from the perspective of detecting faults.

One of the key disadvantages of mutation testing is its high computational cost. A study has shown that approximate transformations can be used to run tests on the approximated version of the mutant set, which has resulted in faster and comparable (and sometimes better) mutation score [23].

D. McCabe Cyclomatic Complexity

The increasing costs of software development have led the research community to come up with a mathematical technique for quantification of modularization so that software modules that are difficult to test or maintain can be identified [24]. Since a direct correlation between program length and structural complexity could not be observed, the LOC metric was discarded for this purpose. On the other hand, the number of control paths was more appealing as this clearly suggested towards testing effort. McCabe's cyclomatic Complexity (CC) is a measure of the number of linearly independent paths through a program.

Considering a node *n* to be an executable statement in the control flow graph and the edges *e* to be the flow of control between them, the cyclomatic complexity v of a strongly connected graph *G* for which any two nodes *r* and *s* have paths connecting them can be expressed as [25]:

$$v(G) = e - n + 1 \tag{3}$$

The graph of a program is made strongly connected by connecting an END node to the BEGIN node. The above equation can be generalized for a program containing multiple components *p*, where each component has a single entry and exit point, by transforming into a graph *S*, whose complexity *v* is calculated as:

$$v(S) = e - n + 2p \tag{4}$$

We have used Jacoco report's "complexity covered" and "complexity missed" counters to calculate the total complexity of each class in our open source projects. The "complexity missed" refers to the number of test cases that fail to go through a module entirely. Thus, a simple addition of these two counters gives us the total complexity of a class. An alternate proposition was to consider the lines, branches and method counters as representations of nodes, edges and program modules for the calculation of complexity. However, this resulted in negative complexity for multiple classes, thereby proving this calculation technique to be inaccurate. It must be noted that exception handling is ruled out as branches by Jacoco since try/catch blocks are assumed to not increase complexity.

Table I (column CC) clearly shows that complexity increases with each revision, which is easily expected as the software evolves and accommodates new features for every new version.

Since the complexity value is the minimum number of paths that, when combined linearly, produces a set of all possible paths, it gives us the optimum number of test cases that is the supremum of the total set of possible test cases to attain maximum coverage. Complexity measures overcome the language-dependent problems presented by LOC metric. Also, LOC metric is not able to measure software quality until late in its lifecycle. Additionally, test coverage is unable to comment on the structural complexity of a program module. Halstead's measure of complexity [26] merely states that an increase in program size will lead to a corresponding increase in complexity but fails to specify the distinctions between operand and operators, and is incapable of measuring code structure, interactions between modules through inheritance and other object-oriented processes.

E. Code churn

The change in the relative complexity between subsequent revisions of software is an important indicator of the trends of bugs or software quality across the revisions. Code changes in general, and increase in modifications in particular, usually goes hand in hand with higher defects unless the tests reduce defects at a higher rate than fault injection. It must be noted that new code is introduced in response to bug fixes, the introduction of new features and refactoring to local modules. All of the above may result in ripple effects to coupled modules, and the relative complexity of each altered module may change as well.

A system *R* is said to have progressed in terms of complexity if the relative complexity ρ_d of modules *m* that were present in the earlier build *i* but removed prior to the later build *j* is higher than the relative complexity ρ_a of the modules that have been added since the earlier build [27]. The relative complexity ρ_s of modules that remain in both builds is the only constant that remains unchanged. This relation can be expressed as:

$$R^i = \sum_{s \epsilon M_s} \rho_s^i + \sum_{d \epsilon M_d} \rho_d^i \tag{5}$$

$$R^j = \sum_{s \epsilon M_s} \rho_s^j + \sum_{a \epsilon M_a} \rho_a^j \tag{6}$$

The code delta is the difference between the two builds and pertains to the relative complexity measure. The code delta δ of module *m* between build *i* and *j* can be measured in terms of relative complexity ρ, as in:

$$\delta_m^{i,j} = \rho_m^j - \rho_m^i \tag{7}$$

Therefore, code churn $\varkappa$ for module *m* between builds *i* and *j* is measured as:

$$\varkappa_m^{i,j} = \left|\delta_m^{i,j}\right| = \left|\rho_m^j - \rho_m^i\right| \tag{8}$$

The net code churn ∇ of the system consisting of total *M* modules over the same builds *i* and *j* can be calculated as:

$$\nabla^{i,j} = \sum_{s \epsilon M_s} \varkappa_s^{i,j} + \sum_{d \epsilon M_d} \rho_d^i + \sum_{a \epsilon M_a} \rho_a^j \tag{9}$$

We have measured the code churn CCh of the selected open-source projects by first counting the number of files that have been added, modified and deleted between each incremental pair of revisions using cloc. The cumulative churn is then calculated as the summation of this value and the previous cumulative code churn. The version related code churn is found to be the average of these values [28].

Table I (column CCh) shows the cumulative change in files with every revision of the open-source projects.

F. Post-release Defect Density

Software quality is most commonly measured in terms of Defect Density (DD). Defect density is the ratio of the number of defects found post-release by users over the total size of the module in thousands of lines of code (KLOC) [29]. Several studies have attempted to find the average number of defects per KLOC [30, 31] whereas another study has established that program size is a good factor to characterize defects and the tendency of a program module to contain defects [32]. Yet another research claims that the number of bugs in a program may be language-dependent [33]. Therefore, it can be seen that defect density is often used to either act as the independent variable to measure the quality of another metric or to evaluate predictors and factors that may influence the bug density.

We have used cloc [34], which is a general-purpose text-analysis package, to count the total number of code lines. We have used the issue tracking system of our open source projects to count the number of bugs that affect each version.

Table I (column DD) shows the varying defect densities after the release of each corresponding version. Defect density is observed to decrease with every revision, which suggests that coding practices may have improved as the software size increased.

It is highly important to be able to accurately predict the defect density through another metric that is easily observable. This is because post-release defect density cannot be measured unless the software has entered the production stage and is being extensively operated by end users. At that point of the software lifecycle, any refactoring or bug fixes are costly.

TABLE I. METRICS VALUES

Open source project	Version number	SC	BC	MS	CC	CCh	DD
P1	4.0	0.8568	0.77	0.36	6178	1270	0.6362
	4.1	0.8536	0.78	0.4	7126	1573.5	0.2622
	4.2	0.8771	0.81	0.42	7167	1849.5	0.0749
	4.3	0.8776	0.81	0.42	7178	1861.5	0.0298
	4.4	0.8995	0.82	0.44	7363	1991.5	0.0740
P2	1.2	0.7459	0.70	0.65	365	0	4.3913
	1.2.2	0.7232	0.71	0.64	417	61	4.9285
	1.3	0.7576	0.76	0.68	471	117.5	1.8245
	1.4	0.7764	0.76	0.64	470	88.5	0.7120
P3	2.2	0.8793	0.86	0.82	2202	0	0.0154
	2.3	0.8795	0.87	0.83	2211	198	0.0276
	2.4	0.8802	0.87	0.82	2255	361.5	0.1601
	2.5	0.881	0.87	0.83	2448	757	0.0558
	2.7	0.8921	0.86	0.81	2984	1287.5	0.0019

III. CORRELATION ANALYSIS: RESULTS AND DISCUSSION

We have used Spearman's rank correlation coefficient to represent the relationship between two metrics under observation in a numerical form. It is a rank statistic that gives a measure of the strength between two parameters. The other popular correlation statistic is Pearson's correlation coefficient. However, Pearson's coefficient was not used in this case since it calculates the linear relationship between two continuous variables, as opposed to Spearman, which calculates the monotonicity between two variables [35]. Spearman noted that the most important requirement was to be able to represent the observed correlation between two variables in numerical form [36], i.e., if variable A and B share a monotonic relationship, the value of Spearman's correlation coefficient ρ will be +1 and vice versa. A value of $\rho = 0$ infers that A and B are completely uncorrelated. The Spearman correlation value ρ from n samples between any pair of metrics A and B with a difference in their ranks d_i for the i^{th} sample is calculated as:

$$\rho_{A,B} = 1 - \frac{6 \sum d_i^2}{n(n^2-1)} \quad (10)$$

The class level correlations were performed on versions 4.4, 1.4, 2.2 for P1, P2 and P3 respectively. The version level correlations were performed for all selected versions, as mentioned in Section I.

A. Statement coverage and Mutation score

We have received a positive correlation value for all our projects (Table II row SC-MS) in this case. This indicates that an increase in statement coverage is accompanied by an increase in the mutation score. In other words, if a test suite is more thorough in going through every executable statement, it is more likely to be better at detecting defects introduced by modified code or mutants. Related work [37] conforms with this conclusion by demonstrating a perfect linear relationship between statement coverage and mutation score when tested on an open-source project called "Spojo". Fig 1. shows that there is a high density of data points that fall in the category of high mutation score and high statement

coverage. There is also a general increase in mutation score for increasing statement coverage. It must be noted that P1 has a significant number of data points that have variable statement coverage but 0 mutation score. Thus, the test cases may be thorough but fail to make correct assertions during mutation testing.

TABLE II. SPEARMAN CORRELATION VALUE

Metrics pairs	Open source projects		
	P1	*P2*	*P3*
SC-MS	0.4964	0.9735	0.6969
BC-MS	-0.6508	0.7411	0.3186
SC-CC	-0.2549	0.6254	-0.2853
BC-CC	-0.4387	0.5303	-0.1962
SC-DD	-0.9	0.4	-0.1
BC-DD	-0.6750	0.175	0.875

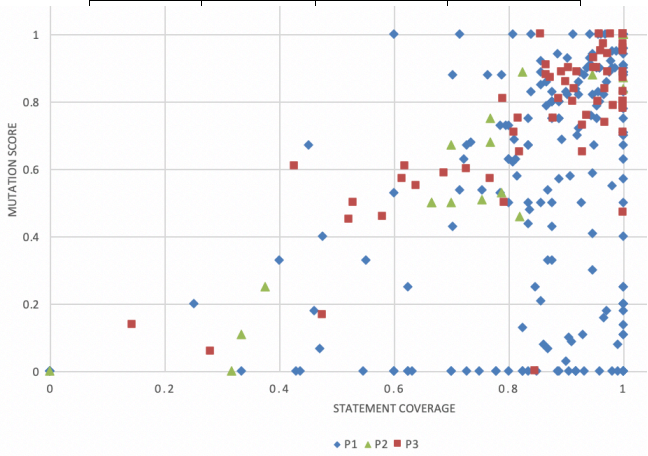

Fig 1. Correlation of statement coverage and mutation score.

B. *Branch coverage and Mutation score*

We have received both positive and negative correlation values for our projects (Table II row BC-MS) in this case. In general, it can be easily justified that test cases that are good at killing mutants, do so through covering every branch and statement. This justification has been proved in the previous correlation between statement coverage and mutation score. However, it can also be argued that the tests that go through every edge in the control flow graph, may not fare so well when the program is mutated, and new edges crop up. It must be noted that the branch coverage values against which the correlation was performed are those obtained by running the test suite against the original code and not with mutants existing in the source code. Also, P1 may show a negative correlation value because almost 33% of the classes had to be unaccounted for due to 0 branches covered and missed counters from the Jacoco report. Additionally, there were some classes for which the mutation score was not generated, e.g., the ones having no line coverage. Fig 2. shows the distribution of mutation score with varying branch coverage. It can be seen that there a lot of classes which has attained 100% branch coverage as well been able to kill all mutants. Although this is a compliment on the test suite in general, it does not help towards finding a conclusive correlation value.

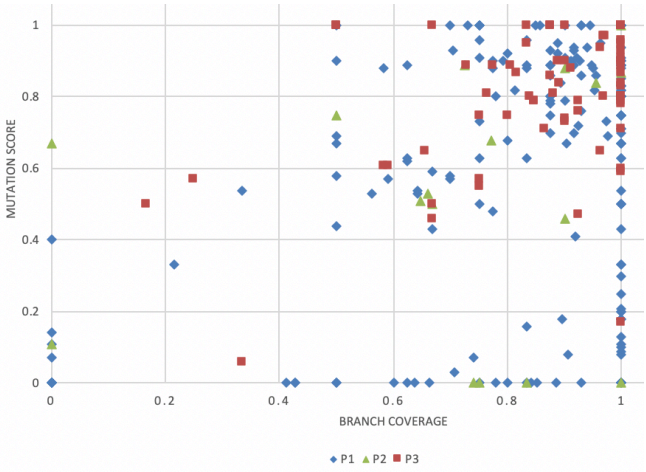

Fig 2. Correlation of branch coverage and mutation score.

C. *Statement coverage and McCabe's Cyclomatic Complexity*

We have received a slight negative correlation value for projects P1 and P3 and a positive value for P2 (Table II row SC-CC) in this case. It must be noted that the negative values are closer to 0, which indicate that there is no direct correlation between statement coverage and complexity. This can be easily justified since an increase in complexity would require the test cases to be more efficient, which is usually not the case. On the other hand, the test cases that are generally designed to be good at coverage will go on exhibiting similar performance for modules of higher complexity. Fig 3. shows a high concentration of classes that have been covered well. However, subjective observation of the graph reveals that there is almost an equal distribution of high and low complexity classes in the ones with good statement coverage, thereby corroborating with the Spearman correlation value.

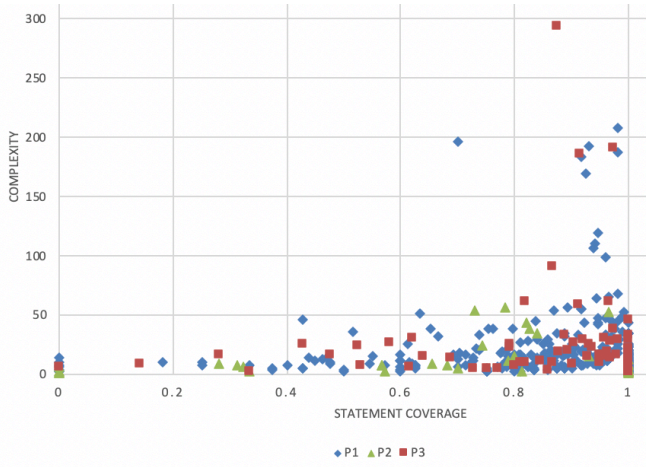

Fig 3. Correlation of statement coverage and McCabe's cyclomatic complexity.

D. *Branch coverage and McCabe's Cyclomatic Complexity*

The justification for the previous correlation follows in this case too as we have received a slight negative correlation value for projects P1 and P3, whereas a positive correlation for P2 (Table II row BC-CC). It must be noted that the projects maintaining a negative correlation in the previous relationship (BC-CC) continue to do so in this case. This suggests that the test cases of P2 were designed to cover more statements and branches even for high complexity modules, whereas those of P1 and P3 fail to gain sufficient coverage for the high complexity modules. Fig 4. again, shows a high concentration of classes with good branch coverage, but there is no marked linear relationship observed between branch coverage and complexity. Additionally, it can be observed that P2 has fewer modules and the average complexity is far less than P1 and P3 (from Table II) – this could be the reason for the test cases being able to gain good coverage on the modules, irrespective of complexity. It was established by a study [38] that larger modules are more prone to errors than smaller modules with relatively less complexity. It can be derived from this finding, that it is easier for the same standard of test suite to cover more statements and branches in the smaller module. Furthermore, the same study showed that the error-prone modules do not necessarily have a higher complexity – it could simply be the quality of the test suite.

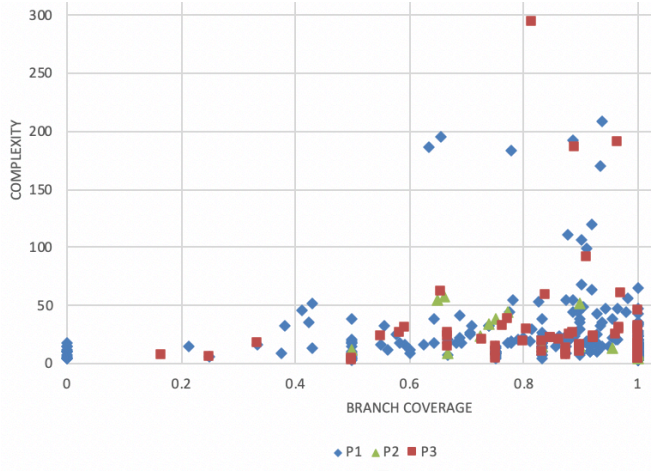

Fig 4. Correlation of branch coverage and McCabe's cyclomatic complexity.

E. *Statement coverage and Defect density*

We see a high negative value of correlation in P1, followed by negative values for P3 as well (Table II row SC-DD). This indicates that test cases with low statement coverage result in a higher number of post-release defects. Fig. 5. shows that with the increase of statement coverage, defect density is usually decreasing with a few exceptions. These exceptions may arise if the size of the software decreases, reminding us that the number of defects is not the only factor governing defect density. It must be noted that the number of samples used in this correlation is extremely low as the number of versions taken into account for each project is not more than 5. Therefore, it would be recommended to not base any conclusions unless bigger sample size is achieved. Also, P2 was not visualized in this case because the defect density was so high that the trends in other values were not visible. But it is apparent from Table I that defect density decreased with an increase in statement coverage.

F. *Branch coverage and Defect density*

We observe contradicting correlation values (Table II row BC-DD) across the projects. P1 exhibits a negative correlation that indicates that with the exception of a few versions, low branch coverage leads to more defects. On the other hand, P3 gives us a positive correlation value. This may be explained by looking at the trends of branch coverage and defect density values across versions of P3 from Table 1 – the branch coverage values remain almost consistent varying from 0.86 to 0.87 whereas the defect density increases initially, possibly due to low maintenance throughout code evolution. However, the defect density drops after version 2.4, and this may be because of a heavy increase in software volume. Fig. 6. shows the variation of defect density and branch coverage across versions. The positive correlation value for P2 can also be attributed to the spike in defect density from the second to third data point (along the X axis), specifically from version 2.3 to 2.4. This could happen due to a major upheaval of code and introduction of new features in this release that led to an increased number of post-release defects that remained undetected by the test suite, even though the statement coverage was maintained almost consistent (if not improved).

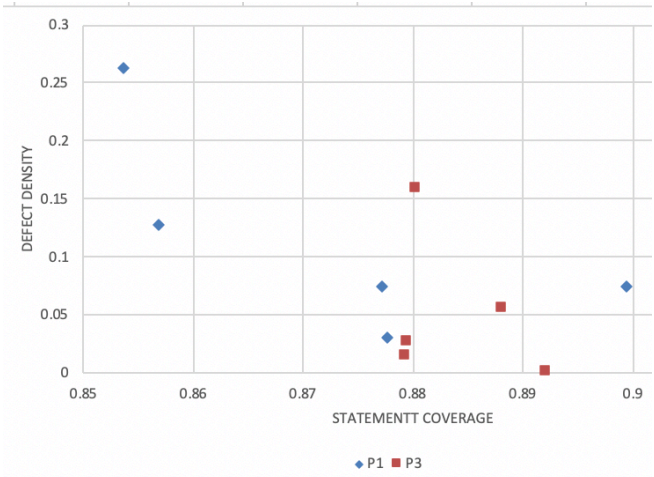

Fig 5. Correlation of statement coverage and Post-release defect density.

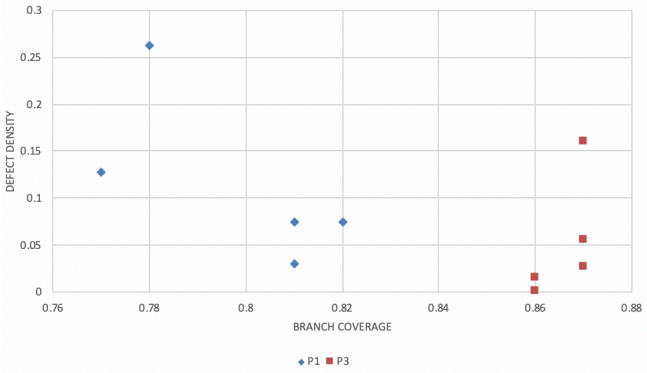

Fig 6. Correlation of branch coverage and Post-release defect density.

G. Code churn and Defect density

We observe a negative correlation value for the selected projects (Table II row CCh-DD). This could be because the size of the software increases with every revision, so much so that the number of defects is less as compared to the lines of code. Although studies [39] show that large and recent changes are accompanied with fault potential, it is difficult to generalize the conclusions of the experimented studies because the same study could have arrived at a different conclusion if it was conducted in an environment beyond its specifications. Fig 7. shows the variation of code churn and defect density across versions.

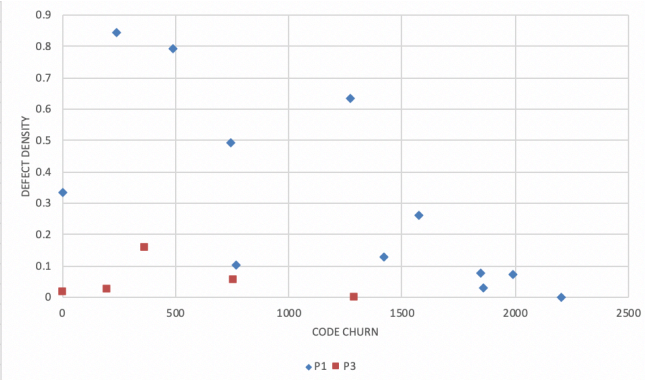

Fig 7. Correlation of code churn and Post-release defect density.

IV. Conclusion

This study was focused on studying background research about related works on software metrics to gain a better understanding of the concepts, measure six popular metrics on three commonly used open-source projects and analyze their correlations. It was also understood that a deeper analysis with more data and better integration among the analyzers could have led to a more valuable study. Future work in this field involving the prediction of software quality through measurement of the quantifiable metrics can lead to a breakthrough in the software industry

Acknowledgment

We sincerely thank our professor Dr. Jinqiu Yang for introducing us to the interesting concepts of software measurement and our teaching assistant Zishuo Ding for clearing our doubts throughout the progress of this project.

References

[1] C. Jones, Applied software measurement: assuring productivity and quality. Emeryville, CA: McGraw-Hill/Osborne, 2008.

[2] N. E. Fenton and M. Neil, “Software metrics: successes, failures and new directions,” *Journal of Systems and Software*, vol. 47, no. 2-3, pp. 149–157, 1999.

[3] J.-P. Jacquet and A. Abran, “From software metrics to software measurement methods: a process model,” *Proceedings of IEEE International Symposium on Software Engineering Standards*.

[4] N. Fenton and M. Neil, “A critique of software defect prediction models,” *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, 1999.

[5] M. Zhao, C. Wohlin, N. Ohlsson, and M. Xie, “A comparison between software design and code metrics for the prediction of software fault content,” *Information and Software Technology*, vol. 40, no. 14, pp. 801–809, 1998.

[6] C. D. Team, *Collections – Home*. [Online]. Available: http://commons.apache.org/proper/commons-collections/. [Accessed: 10-Apr-2020].

[7] M. Cooper, *FileUpload – Home*. [Online]. Available: http://commons.apache.org/proper/commons-fileupload/. [Accessed: 10-Apr-2020].

[8] C. D. Team, *Commons IO – Commons IO Overview*. [Online]. Available: http://commons.apache.org/proper/commons-io/. [Accessed: 10-Apr-2020].

[9] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software unit test coverage and adequacy. ACM Comput. Surv. 29, 4 (December 1997), 366–427. DOI:https://doi.org/10.1145/267580.267590

[10] “JaCoCo Java Code Coverage Library,” *EclEmma*, 14-Jan-2020. [Online]. Available: https://www.eclemma.org/jacoco/. [Accessed: 10-Apr-2020].

[11] “The influence of size and coverage on test suite effectiveness.” [Online]. Available: https://www.researchgate.net/publication/220854552_The_influence_of_size_and_coverage_on_test_suite_effectiveness. [Accessed: 10-Apr-2020].

[12] *Software reliability growth with test coverage - IEEE Journals & Magazine*. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1044339. [Accessed: 10-Apr-2020].

[13] J. D. Musa, *Software Reliability Engineering*. Place of publication not identified: IEEE Computer Society Press, 2004.

[14] T. T. Chekam, M. Papadakis, Y. Le Traon and M. Harman, "An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption," *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, Buenos Aires, 2017, pp. 597-608.

[15] A. Panichella, F. M. Kifetew and P. Tonella, "Reformulating Branch Coverage as a Many-Objective Optimization Problem," *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, Graz, 2015, pp. 1-10.

[16] Y. Wei, B. Meyer, and M. Oriol, “Is Branch Coverage a Good Measure of Testing Effectiveness?,” *Empirical Software Engineering and Verification Lecture Notes in Computer Science*, pp. 194–212, 2012.

[17] P. Piwowarski, M. Ohba and J. Caruso, "Coverage measurement experience during function test," Proceedings of 1993 15th International Conference on Software Engineering, Baltimore, MD, USA, 1993, pp. 287-301.

[18] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," in IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 649-678, Sept.-Oct. 2011.

[19] P. Chevalley, "Applying mutation analysis for object-oriented programs using a reflective approach," Proceedings Eighth Asia-Pacific Software Engineering Conference, Macao, China, 2001, pp. 267-270.

[20] D. Tengeri et al., "Relating Code Coverage, Mutation Score and Test Suite Reducibility to Defect Density," 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Chicago, IL, 2016, pp. 174-179.

[21] “Real world mutation testing,” *PIT Mutation Testing*. [Online]. Available: https://pitest.org/. [Accessed: 10-Apr-2020].

[22] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, “PIT: a practical mutation testing tool for Java (demo),” *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, 2016.

[23] M. Gligoric, S. Khurshid, S. Misailovic and A. Shi, "Mutation testing meets approximate computing," 2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER), Buenos Aires, 2017, pp. 3-6.

[24] T. J. McCabe. 1976. A Complexity Measure. IEEE Trans. Softw. Eng. 2, 4 (July 1976), 308–320. DOI:https://doi.org/10.1109/TSE.1976.233837

[25] M. Shepperd, "A critique of cyclomatic complexity as a software metric," in Software Engineering Journal, vol. 3, no. 2, pp. 30-36, March 1988.

[26] M. H. Halstead, *Elements of software science*. New York: North Holland, 1979.

[27] J. Munson and S. Elbaum, “Code churn: a measure for estimating the impact of code change,” *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*.

[28] C. Faragó, P. Hegedűs and R. Ferenc, "Cumulative code churn: Impact on maintainability," 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM), Bremen, 2015, pp. 141-150.

[29] Ronald B. Finkbine. 1996. Metrics and Models in Software Quality Engineering. SIGSOFT Softw. Eng. Notes 21, 1 (January 1996), 89. DOI:https://doi.org/10.1145/381790.565681

[30] Akiyama, F. (1971). An Example of Software System Debugging. *IFIP Congress*.

[31] Chulani, Sunita & Boehm, Barry. (1999). Modeling software defect introduction and removal: COQUALMO (COnstructive QUALity MOdel).

[32] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," in IEEE Transactions on Software Engineering, vol. 26, no. 8, pp. 797-814, Aug. 2000.

[33] G. Phipps, “Comparing observed bug and productivity rates for Java and C ,” *Software: Practice and Experience*, vol. 29, no. 4, pp. 345–358, Oct. 1999.

[34] “Overview,” *CLOC*. [Online]. Available: http://cloc.sourceforge.net/. [Accessed: 10-Apr-2020].

[35] J. Hauke and T. Kossowski, “Comparison of Values of Pearsons and Spearmans Correlation Coefficients on the Same Sets of Data,” *Quaestiones Geographicae*, vol. 30, no. 2, pp. 87–93, Jan. 2011.

[36] C. Spearman, “The Proof and Measurement of Association between Two Things,” *The American Journal of Psychology*, vol. 15, no. 1, p. 72, 1904.

[37] B. Assylbekov, E. Gaspar, N. Uddin and P. Egan, "Investigating the Correlation between Mutation Score and Coverage Score," 2013 UKSim 15th International Conference on Computer Modelling and Simulation, Cambridge, 2013, pp. 347-352.

[38] Basili VR, Perricone BT. Software errors and complexity: an empirical investigation. *Communications of the ACM.* 1984;27(1):42–52.

[39] Graves, T. L., Karr, A.F., Marron, J.S., Siy, H., "Predicting Fault Incidence Using Software Change History," IEEE Transactions on Software Engineering, vol. 26, pp. 653-661, 2000