Concordia University

*Department of Computer Science and Software Engineering*

*SOEN 6611 Report*

*Summer 2019*

Metrics and Correlation Analysis

Jiahui Wang – 40070981 – wjhmars@gmail.com

Lei Xu – 40046329 – ethanxu90@gmail.com

Chongwen Li – 40042828 – lichongwen1994@gmail.com

Zhongxu Huang – 40052560 – realdonald9@gmail.com

Xiayan Zhong – 40051749 – liangliang44@hotmail.com

Metrics and Correlation Analysis

*Abstract*—Nowadays, the ever-increasing complexity of software, together with greater competitive pressures and skyrocketing costs, has pushed the need of the software testing to new heights. Therefore, metrics, an important indicator of the efficiency and effectiveness of a software testing process, are gaining importance and acceptance by providing quantitative approach to the development and validation during the whole software development life cycle. In addition, metrics can help developers obtain the necessary information that they need to keep improving software products, reduce errors and achieve desired goal. In the report, we will focus on six metrics from four aspects to describe their definitions, algorithms and calculation and then analyze correlations between them through several experiments. Also, the procedure of experiments and conclusion will be discussed.

Keywords—software testing, software quality, metrics, coverage, effectiveness, complexity, mutation, Halstead, maintenance, defect density

# Introduction

In recent times, the growth of software has increased manifolds. All kinds of software products are developed for companies as well as for individuals. With the increase in the software’s availability, the attention has transferred to the software testing which is mainly about the software quality evaluation and enhancement. The software testing technologies, merging as a dominant software engineering practice, do help in several aspects, like effective cost control, quality improvement and risk reduction etc. A key of the research field has been ‘measurement and metrics’ of the software testing.

Software Metrics is the way to measure and monitor test activities. More importantly, they give insights into the team’s test progress, productivity, and the quality of the system under test. When we ask ourselves “What have we tested?”, metrics will give us better answers than just “we have tested it.” Different teams measure various aspects depending on what they want to track and control or improve. The purpose of the project is to analyze the six following selected metrics from five aspects of software testing:

1. Test Coverage Metrics – Statement Coverage & Branch Coverage
2. Test Suite Effectiveness Metrics – Mutation score
3. Complexity Metrics – Cyclomatic Complexity
4. Software Maintenance Effort Metrics – Code Churn
5. Software Quality Attribute Metrics – Post-release Defect Density

And then we will present comprehensive descriptions and relationships between the selected metrics.

In summary, the main contributions of the project are as follows:

* Detailed Descriptions of six metrics from five aspects, like their definitions and algorithms.
* Comprehensive survey of previous studies on these metrics, including correlations.
* Experiment Implementation on five selected programs to analyze the six metrics
* In-depth analysis on the correlations between these metrics

1. Investigated Subject programs

Non-commented Line of Code (NLOC) is reported for the most recent version. LOC refers to non-comment, non-blank lines of code and was measured with sloccount

|  | Subject Programs | | | | |
| --- | --- | --- | --- | --- | --- |
| **Criterion** | Apache Commons Configuration | Apache Commons Collection | Apache Commons Digester | Apache Commons JXpath | Apache Commons Codec |
| URL | http://commons.apache.org/configuration/ | http://commons.apache.org/collections/ | http://commons.apache.org/proper/commons-digester/ | http://commons.apache.org/proper/commons-jxpath/ | http://commons.apache.org/proper/commons-codec/ |
| Bug-tracking | http://issues.apache.org/jira/browse/CONFIGURATION | http://issues.apache.org/jira/browse/COLLECTIONS | http://issues.apache.org/jira/browse/DIGESTER | http://issues.apache.org/jira/browse/JXPATH | http://issues.apache.org/jira/browse/CODEC |
| SLOC | 149,713 | 125,213 | 49,183 | 43,911 | 38,395 |
| Version | 2.5 | 4.3 | 3.2 | 1.3 | 1.12 |
| Language | Java | Java | Java | Java | Java |

The remainder of the project report is structured as follows. Section II describes the related previous works of the selected six metrics. Section III describes our methodology and experiments we performed to analyze correlations. And Section IV describes the conclusions.

# Related Work

## Coverage Metrics

It is undeniable that test coverage is of importance indicator of software quality and software maintenance. It helps in evaluating the effectiveness of testing by providing data on different coverage items. Test coverage was among the first methods invented for systematic software testing. The first published reference was by Miller and Maloney in Communications of the ACM in 1963 [1]. The idea of using criteria for testing concurrent software products was originally proposed by Takahashi et al. [2]. The concurrent coverage criteria intends to find concurrent software specific defects, such as race conditions. Gupta and Jalote [3] found a way to use mutation analysis to experimentally evaluating effectiveness and efficiency of coverage criteria for testing. In addition, Whalen et al. [4] defined test coverage metrics on high-level formal software requirements to support structural or white box testing. Worth mentioning, Jose C. Costa and Jose C. Monteiro et al. [5] presented a new program execution based approach to generate input data for branch coverage.

To conclude, as far as we are aware test coverage metrics appeared earliest, and is widely used by developers nowadays, especially statement coverage and branch coverage.

## Test Suite Effectiveness Metrics

A test suite quality is often measured by the number of bugs it can find (aka. kill). Previous studies have analyzed the quality of a test suite by its ability to kill mutants, i.e., artificially seeded faults. DeMillo et al. [6] and Hamlet [7] are the first two people who proposed the idea of using mutants to measure test suite adequacy. Offutt and others [8] explored the idea extensively and showed empirical support for one of the basic premises of mutation testing [9]. In addition, A lot of work have been done on the subject of code coverage and test suite effectiveness. An example is a study written by A. S. Namin and J. H. Andrews [10] where they explored the relationship between three properties of test suites: coverage, size and fault-finding effectiveness. In another study made by L. Inozemtseva and R. Holmes [11], they evaluated the relationship between coverage, the size of a test suite and the test suite effectiveness for large Java projects. The results indicates that there is a low to moderate correlation between coverage and test suite effectiveness when the number of tests in the test suite are controlled for.

It concludes that mutation testing is a means of creating more effective test cases, thus mutants can be applied in the measurement of test suite effectiveness. Additionally, it is also worth studying the relationship between coverage and test suite effectiveness.

## Complexity Metrics

LOC metric is considered as a traditional metric which counts the number of lines form the source code and it was not considered as an adequate metric because if a source code contains 500 lines consisting of 100 decision statements and which may contain million of paths then with LOC metric it is possible to measure and test only small proportion [12]. The software complexity metric is one of the measurements that use some of the internal attributes or characteristics of software to know how they effect on the software quality. In 1977, Halstead’s measure of complexity was purposed which was used to calculate the software complexity by counting the number of unique operators and operands [13] but it ignores the complexity from the control flow graph and put the same emphasis if source code contain operands and operators with branches or not [14]. So, for that McCabe’s cyclomatic complexity metric was purposed which includes the complexity from the control graph. But McCabe’s cyclomatic complexity metric failed to measure the exact software complexity if the interaction between different classes is higher. So, the concept of object coupling needs to be included in the concept of cyclomatic complexity. Coupling factor also needs to be introduced in McCabe cyclomatic complexity which lags behind as it does not consider basic elements like class, polymorphism, encapsulation etc. [15]

To summarize, we, for now, are aware that McCabe’s cyclomatic complexity metric was purposed to include the complexity from the control graph. Therefore, the cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program.

## Software Maintenance Effort Metrics

If effort estimates are not assessed easily upfront by software maintainers, serious problems may be occurred in large maintenance projects, or when engineers make repeated maintenance changes to software. A considerable amount of literature has investigated different types of software maintenance measurement: corrective maintenance effort [19], adaptive maintenance effort [20], maintainability index [21], maintenance time[22] and code churn [23]. Code churn has been introduced by Elbaum and Munson in 1998 [24]. They define it as the difference between two versions of the same system, as a sum of the added, modified and deleted lines. Alija and Dumitrescu use code churn as one of their key metrics to understand product line evolution [25]. They use the change metrics to determine whether it is safe to release a new version. On top of that, Alija and Dumitrescu mention briefly that for the systems they studied, which are industrial, the majority (88%) of code churn is caused by new lines of code. Ostrand et al. [26] use information of file status such as new, changed, unchanged files along with other explanatory variables such as lines of code, age, prior faults etc. as predictors in a negative binomial regression equation to predict the number of faults in a multiple release software system.

In conclusion, we, after studying previous related work, are aware that the code churn is a measure of the amount of code changes taking place within a software unit over time. It is easily extracted from a system’s change history, as recorded automatically by a version control system. Most version control systems use a file comparison utility (such as diff) to automatically estimate how many lines were added, deleted and changed by a programmer to create a new version of a file from an old version. These differences are the basis of churn measures.

## Software Quality Attribute Metrics

Quality in use metrics identifies the metrics used to measure the effects of the combined quality characteristics for the user. Our efforts build upon the main reference works by Fenton and Bieman [28] in the area of software metrics and by Kan [29] in the more specific area of software quality metrics. Defect density and related metrics have been receiving considerable attention. Gupta et al. [30] conducted a longitudinal case study of two related internal projects, a reusable framework and its application, and two metrics, defect density and change density, with defect density used as a quality metric. Shah, Morisio and Torchiano [31] carried out a metastudy that cites 19 other papers studying 109 projects and determined that further work was needed to identify effective predictors of defect density.

Drawing general conclusions from empirical studies, we beware that post-release defect density defines a quality indicator for the software quality. It is possible to measure only the identified post-release defects. But it will not be always correct to compare only the number of post-release defects. It can also be argued that the complexity may affect the number of identified post-release defects [32]. Hence, our project will mainly focus on the data collection and universal standard to measure the post-release defect density of selected open source systems.

# Experiment Material and Procedure

In this section, we describe the experiment we performed, the subject programs and the artifacts with which they are associated. At the end of the section, we will also discuss the relationships between them.

## Methodology

Our goal of the final project is first to obtain comprehensive knowledge of the six selected metrics from five aspect, like its definition and algorithm e.g., then to measure these metrics on the selected five open-source systems, and finally analyze correlations between them. To accomplish this, we performed the following high-level steps:

1. Study the previous related work regarding these six metrics to learn their definitions, algorithms and measurements (Section II)
2. Select five subject programs that meet the requiements
3. Select tools to measure metrics
4. Analyze correlations using the spearman.

## Subject Programs

Table 1 lists the 5 subject programs we used in our experiments. These programs satisfy the following desiderata:

1. Each program has more than 100k lines of code, enabling us to better complete the measurement covering selected metrics.
2. Each program has a version control and bug-tracking system, enabling us to locate real faults.
3. Each program is released with a comprehensive test suite, enabling us to experiment with developer-written test suites apart from automatically-generated ones

## Implementation

In the project, we used SLOC tool to calculate the statement line of code for selecting suitable systems. Through comparing the number of lines of each system, we eventually decided to use Apache-Commons-collections (125K), Apache-Commons-configuration (149K), Apache-Commons-digester (49K), Apache-Commons-JXpath (43K), and Apache-Commons-Codec (38K) these five systems as our experimental subjects.

1. *Branch Coverage, Statement Coverage and Cyclomatic Complexity*

Both of branch coverage and statement coverage are intuitive metric for measuring test coverage capacity. We applied Jacoco plugins in each of these five systems’ .pom file and plugin block to generate the Jacoco reports.

In the generated Jacoco reports, we can directly get branch coverage by calculating the percentage of the branch covered (branch covered / branch covered + branch missed), and statement coverage (line covered / line missed + line covered). Cyclomatic complexity is used to indicate the complexity of a program. We can also calculate it from Jacoco report by complexity missed plus complexity covered. Finally, all of these three metrics are class level.

1. *Mutation Score*

Mutation score is an important metric for measuring test effectiveness of test suits. As same for generating Jacoco report, we applied PItest plugin in .pom file. From the generated PItest report, we can obtain mutation score for class level data.

1. *Code Churn*

Code Churn is a metric which measures the volume of change a system went through between two moments in time. Since the fifth metrics is to measure the software maintenance effort, we believe the metric closest to code changes, churn, is the most applicable.

To compare the churn numbers, we first need a uniform way of calculating churn. For the experiment, we used the git *diff* command [30]. Using this, it is possible to give two SHA1 hashes pointing to versions of the systems, and getting the number of added and deleted lines between those versions. As such, for this experiment we only detect changes and additions. An example is shown as follows:

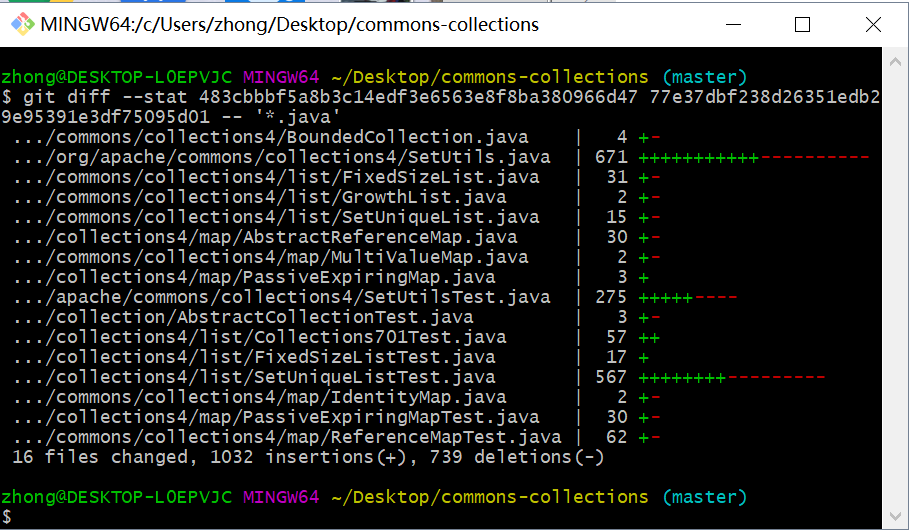


Figure 1: Code Churn

1. *Post-release Defect Density*

Post-release Defect Density (also called operational defects) defines a quality indicator for the software quality. It is the number of identified defects found during the operational phase. Only the total number of added or modified source lines of code are used by the calculation of the post-release Defect Density. The definition of Post-release Defect Density is shown as follows:



Note that is the total number of post-release defects. And KSLOC is the total number of added or modified SLOC, where SLOC is the total number of 1000 source lines of code.

For this experiment, we firstly used the result of code churn to get the changes of code, which is KSLOC in different versions in the project. Then, we used the issue tracking system (e.g. see the figure 2) to search the project we would like to use in the experiment. The next step is to use the filter by setting conditions to find bugs. As such, we, in the experiment, set the issue type as bug and the version which bug affected. Also we knew the time when the bug was found based on the sorting order. At last, we calculated the data collected according to the formula above. It is worth mentioning that the bug report can be downloaded if needed (e.g. see the figure 3).

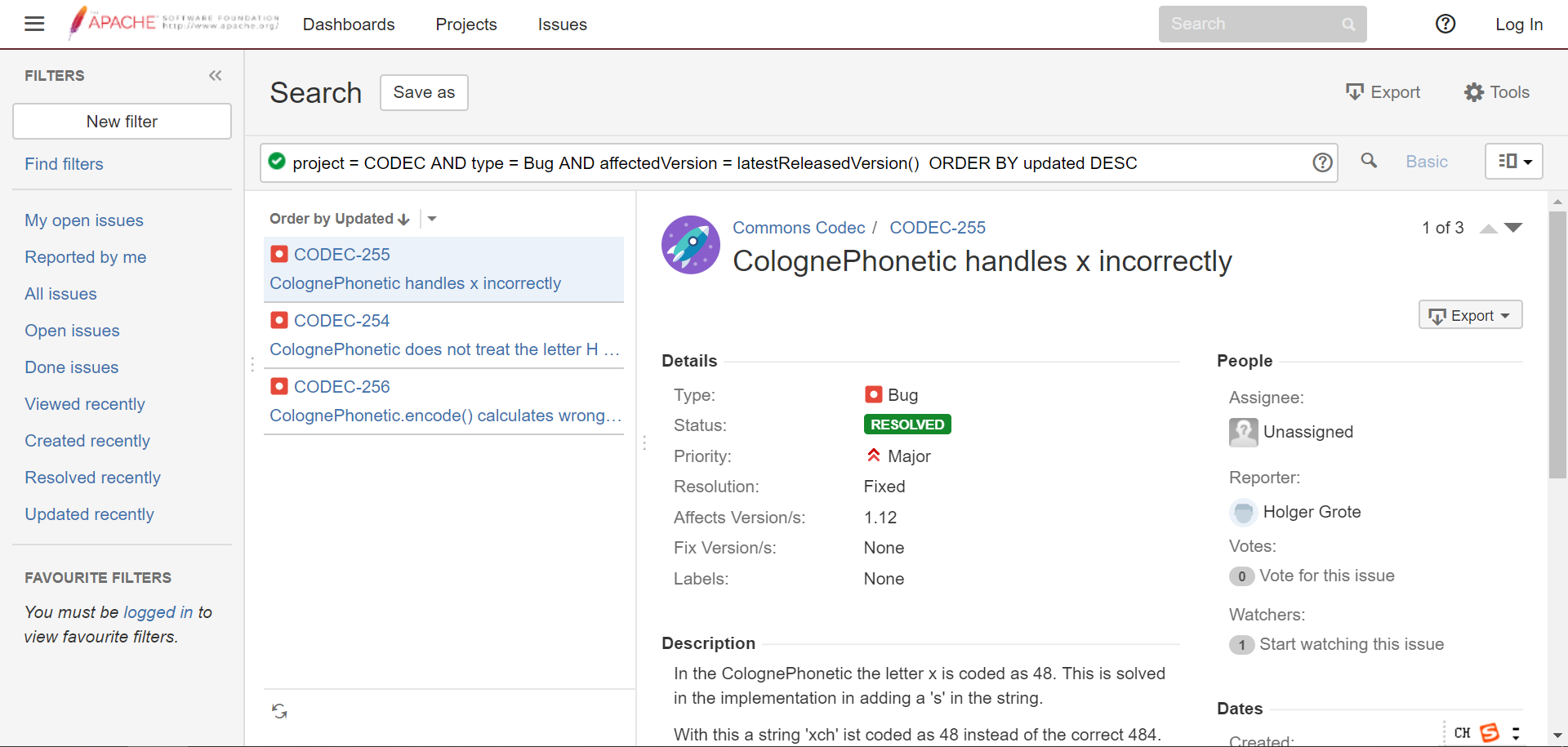


Figure 2: Issue-tracking System

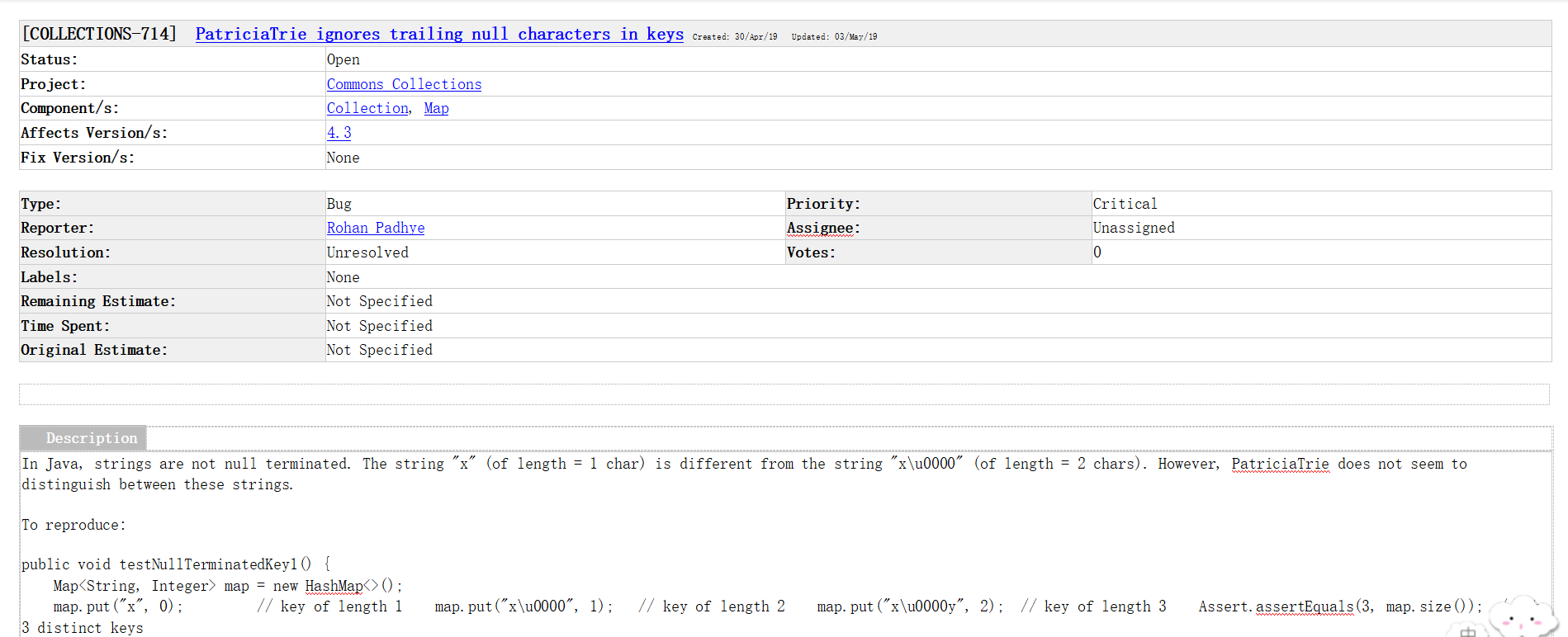


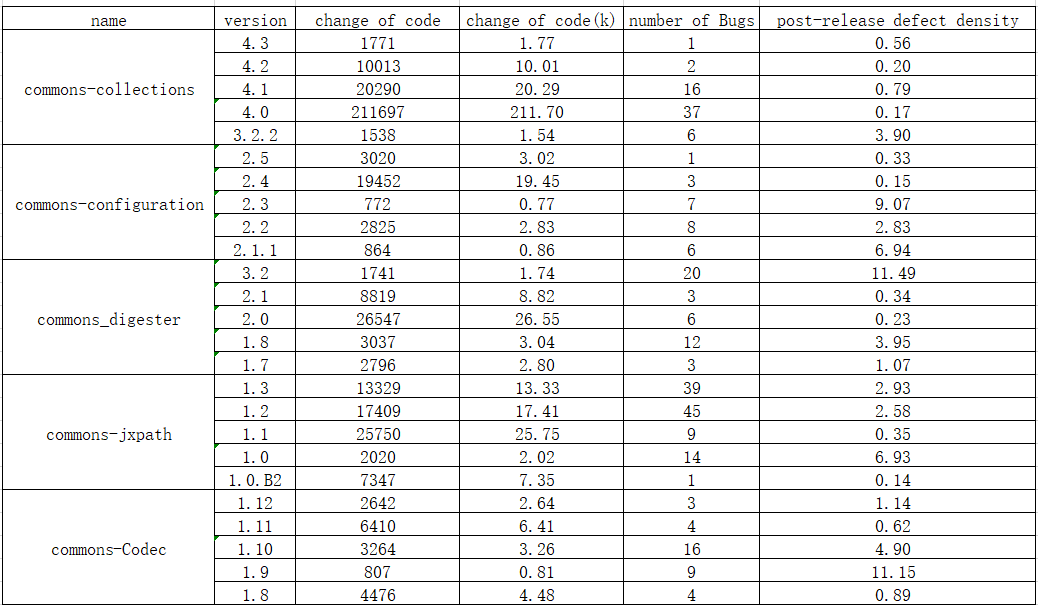
Figure 3: Bug Report

## Results

We first present the results we obtained from the implementations above, followed by the correlations between those metrics.

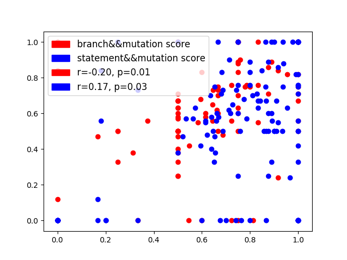
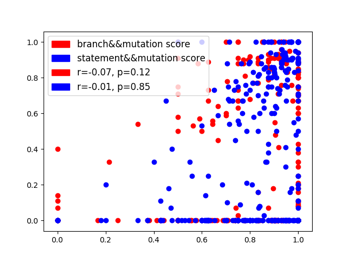
Since the result data from Jacoco and PItest are too big to display here, we, in the section, only present the results of metric 5 and metric 6. It shows as follows:

1. Result of Metric 5 & 6



Each part of correlation graphs contains five graphs. The graphs represent collections project, configuration project, digester project, codec project and jxpath project from left to right respectively. Each dot represents a test suite.

Figure 4 shows the correlation between coverage metrics and test suite effectiveness metric. And the red color represents the correlation between branch coverage metric and mutation score metric, while the blue represents



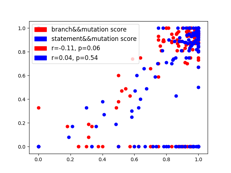
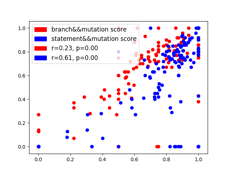
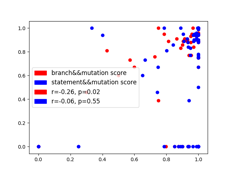


Figure 4: Correlation between Metric 1&2 and Metric 3

the correlation between statement coverage metric and mutation score metric.

Figure 5 shows the correlation between coverage metrics and test suite effectiveness metric. And the red color represents the correlation between branch coverage metric and cyclomatic complexity metric, while the blue represents the correlation between statement coverage metric and cyclomatic metric.

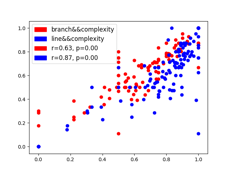
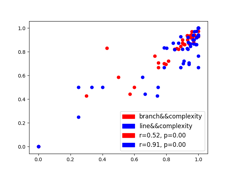
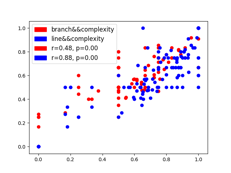
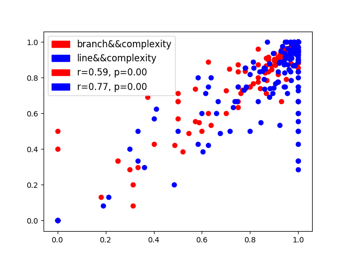
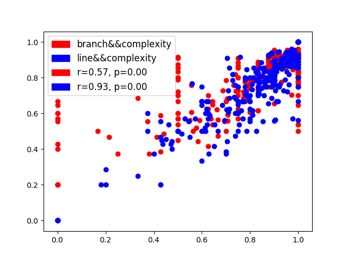


Figure 5: Correlation between Metric 1&2 and Metric 3

Figure 6 shows the correlation between coverage metrics and software quality attribute metric. And the red color represents the correlation between branch coverage metric and post-release defect density metric, while the blue represents the correlation between statement coverage metric and post-release defect density metric.

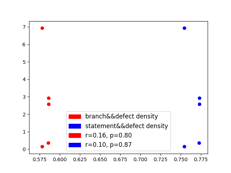
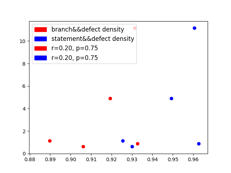
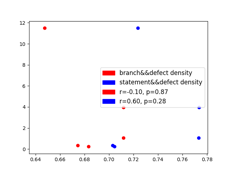
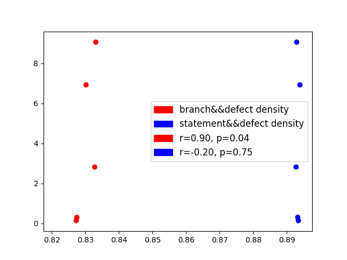
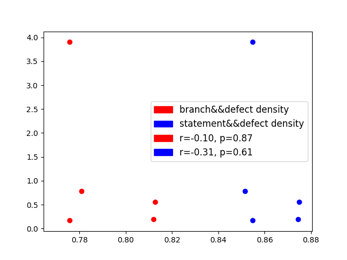


Figure 6: Correlation between Metric 1&2 and Metric 6

Figure 7 shows the correlation between software maintenance effort metrics and software quality attribute metric.

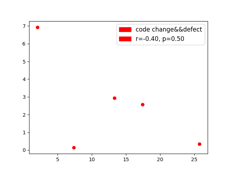
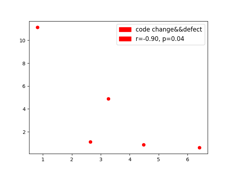
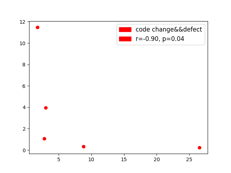
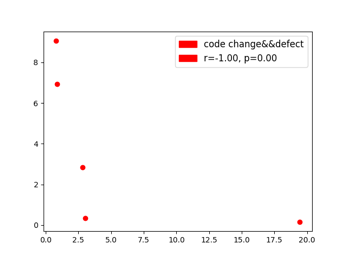
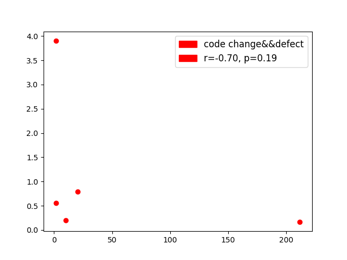


Figure 7: Correlation between Metric 5 and Metric 6

## Analysis

After collecting data, we need to analyze correlations between these metrics. The correlation coefficient is a measure of linear association between variables. So, we here use spearman, one of correlation coefficients, to judge how strong the two metrics are related. We decided to use python and java to analyze the data we obtained.

1. *Correlation between each coverage metrics (Metric 1&2) and test suite effectiveness (Metric 3)*

From the table III, we can see that, for the metric 1 and metric 3, although they are all correlated, the correlation is very weak. The result of Metrics (2&4) shows the same picture. But, for the project (Jxpath), the correlation between statement coverage and mutation score is strong. The overall correlation is still very weak. Hence, we come to a conclusion that there is a low to moderate correlation between coverage and effectiveness when the number of test cases in the suite is controlled for. In addition, we found that stronger forms of coverage do not provide greater insight into the effectiveness of the suite.

1. Correlation between Metrics 1,2 - 3

|  |  |  |
| --- | --- | --- |
| Branch Coverage & Statement Coverage VS Mutation Score (Class Level) | | |
| System Name | Spearman (BC) | Spearman (SC) |
| Collections | 0.07 | 0.01 |
| Configurations | 0.11 | 0.04 |
| Digester | 0.2 | 0.17 |
| Codec | 0.26 | 0.06 |
| Jxpath | 0.23 | 0.61 |

1. *Correlation between each coverage metrics (Metric 1&2) and complexity metrics (Metric 4)*

According to the table IV, we can conclude that, for the metric 1 and metric 4, all of them are positively correlated, only one of them is strong correlated and the rest are moderately correlated. For the metric 2 and metric 4, the overall correlation is the same. But it is worth mentioning that they are all strong correlated, especially two of them are very strong correlated. Therefore, we can infer that classes with higher complexity are high likely to have high coverage test suites.

1. Correlation between Metrics 1,2 - 4

|  |  |  |
| --- | --- | --- |
| Branch Coverage & Statement Coverage VS Cyclomatic Complexity (Class Level) | | |
| System Name | Spearman (BC) | Spearman (SC) |
| Collections | 0.57 | 0.93 |
| Configurations | 0.59 | 0.77 |
| Digester | 0.48 | 0.88 |
| Codec | 0.52 | 0.91 |
| Jxpath | 0.63 | 0.87 |

1. *Correlation between each coverage metrics (Metric 1&2) and software quality attribute metrics (Metric 6)*

It can be seen from Table V that except the configurations project, the correlation between branch coverage and post-release defect density is very weak. The same picture shows in the correlation between statement coverage and post-release defect density except the digester project this time. Overall, we conclude that high test coverage is not sufficient measure to find bugs. On the contrary, test coverage more accurately gives a measure of the extent to which the code has not been tested. This means that if we have a low test coverage metric, then we can be sure that there are significant portions of our code that are not tested, which means that there may be a lot of bug not found. The inverse however is not necessarily true. Having high test coverage is not a sufficient indicator that our code has been sufficiently tested.

1. Correlation between Metrics 1,2 - 6

|  |  |  |
| --- | --- | --- |
| Branch Coverage & Statement Coverage VS Post-release Defect Density (Version Level) | | |
| System Name | Spearman (BC) | Spearman (SC) |
| Collections | 0.1 | 0.31 |
| Configurations | 0.9 | 0.2 |
| Digester | 0.1 | 0.6 |
| Codec | 0.2 | 0.2 |
| Jxpath | 0.16 | 0.1 |

1. *Correlation between software maintenance effort metrics (Metric 5) and software quality attribute (Metric 6)*

As seen from the table below, it shows that the correlation between code churn and post-release defect density is very strong correlated except the coefficient in project – Jxpath, which is moderately correlated. Based on it, it concludes that the more the code changed in the current version, the more bugs it will bring in the next version.

1. Correlation between Metrics 5 - 6

|  |  |
| --- | --- |
| Code Churn VS Post-release Defect Density (Version Level) | |
| System Name | Spearman |
| Collections | 0.7 |
| Configurations | 1.0 |
| Digester | 0.9 |
| Codec | 0.9 |
| Jxpath | 0.4 |

# Conclusion

To sum, we, in theis project, firstly selected six metrics, named branch coverage, statement coverage, mutation score, cyclomatic complexity, code churn and post-release defect density respectively, from five aspects. Then we chose five open source system to perform experiments to measure those metrics. After collecting data, we tried to analyze correlations between them using the spearman correlation coefficient. And we found that:

1. Stronger forms of coverage do not provide greater insight into the effectiveness of the suite.
2. Classes with higher complexity are high likely to have high coverage test suites.
3. High test coverage is not sufficient measure to find bugs.
4. The more the code changed in the current version, the more bugs it will bring in the next version.

Our feature work includes more experiments on various open-source systems written by different kinds of programming languages, like C and C++ etc., and also study other metrics from the five aspects and their correlations.

##### Acknowledgment

Many thanks to the Prof. Jinqiu Yang and Zishuo Ding for valuable suggeations when we stuck in problems and for helping us find right tools to perform the experiment. Also thanks to all the researchers who dived in the study of software measurement. We would also like to thank developers who worked on and improved these subject programs and artifacts over the years.

##### References

1. Joan C. Miller, Clifford J. Maloney, “Systematic mistake analysis of digital computer programs”. Communications of the ACM. New York, NY, USA: ACM. 6 (2): 58–63. doi:10.1145/366246.366248. ISSN 0001-0782.
2. Takahashi, J., H. Kojima, et al. (2008). “Coverage Based Testing for Concurrent Software”, In 28th International Conference on Distributed Computing Systems Workshops, 2008. ICDCS '08.
3. Jalote, A. Gupta. P. (2008). "An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing.", In International Journal on Software Tools for Technology Transfer (STTT), Volume 10, Number 2, 145-160, DOI: 10.1007/s10009-007-0059-5.
4. Whalen, M. W., M. P. E. Heimdahl, et al. (2006). “Coverage metrics for requirements-based testing.”2006, In Proceedings of the 2006 International Symposium on Software Testing and Analysis, ISSTA 2006, pp. 25-35.
5. Jose C. Costa, Jose C. Monteiro (2004), “Input Generation for Branch Coverage in Software Testing”.
6. R. A. DeMillo, R. J. Lipton, F. G. Sayward, “Hints on Test Data Selection: Help for the Practicing Programmer,” IEEE Computer, vol. 11 (4), pp. 34-41, 1978.
7. R. G. Hamlet, “Testing programs with the aid of a compiler,” IEEE Transactions on Software Engineering, vol. 3 (4), pp. 279-290, 1977.
8. A. J. Offutt and R. H. Untch, “Mutation 2000: Uniting the Orthogonal,” Proc. Mutation, San Jose, CA, USA, pp. 45- 55, October, 2000.
9. A. J. Offutt, “Investigations of the Software Testing Coupling Effect,” ACM Transactions on Software Engineering and Methodology, vol. 1 (1), pp. 3-18, 1992.
10. Akbar Siami Namin, James H. Andrews, “The influence of size and coverage on test suite effectiveness”, In Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA ’09, pages 57–68, New York, NY, USA, 2009. ACM.
11. Laura Inozemtseva, Reid Holmes, “Coverage is not strongly correlated with test suite effectiveness”, In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.
12. T. J. Mccabe, “A Complexity Measure”, IEEE Transactions on Software Engineering, Vol. 2, No. 4, pp.308 – 320, 1976.
13. V. Y. Shen, T. Yu and S. M. Thebaut, ”Identifying Error-Prone Software-An Empirical Study”, IEEE Transactions on Software Engineering, Vol. SE-11, No. 4, pp.317 – 324, 1985.
14. S. Yu and S. Zhou, “A Survey on Metric of Software Complexity”, IEEE International Conference on Information Management and Engineering (ICIME), 16-18, Chengdu, pp. 352 – 356, 2010.
15. B. Souley and B.Bata, “A Class Coupling Analyzer for Java Programs”, West African Journal of Industrial and Academic Research, Vol.7, No.1, pp. 3-13, 2013.
16. A. De Lucia, E. Pompella, and S. Stefanucci, "Assessing effort estimation models for corrective maintenance through empirical studies," Information and Software Technology, vol. 47, pp. 3- 15, 1/1/ 2005.
17. F. Fioravanti, P. Nesi, "Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems," IEEE Transactions on Software Engineering, vol. 27, pp. 1062-1084, 2001.
18. S. C. Misra, "Modeling Design/Coding Factors That Drive Maintainability of Software Systems," Software Quality Journal, vol. 13, pp. 297-320, 2005.
19. R. K. Bandi, V. K. Vaishnavi, D. E. Turk, "Predicting maintenance performance using object-oriented design complexity metrics," IEEE Transactions on Software Engineering, vol. 29, pp. 77-87, 2003.
20. Misra, S. C. 2005. “Modeling design/coding factors that drive maintainability of software system”. Software Quality Journal, Vol. 13(3), pp. 297- 320.
21. Tahvildari, L., Gregory, R. Kontogiannis, K.. 1999. “An approach for measuring software evolution using source code features”. Proceedings of the Sixth Asia Pacific Software Engineering Conference, IEEE Computer Society Press, Silver Spring MD, pp.10–17.
22. Hayes, J. H., Zaho, L. 2005. “Maintainability prediction a regression analysis of measures of evolving systems”. Proceedings of 21st IEEE International Conference on Software Maintenance, pp.601- 604.
23. S. Elbaum and J. Munson. Code churn: a measure for estimating the impact of code change. In *International Conference on Software Maintenance*, ICSM ’98, pages 24– 31. IEEE, 1998.
24. S. Elbaum and J. Munson. Code churn: a measure for estimating the impact of code change. In *International Conference on Software Maintenance*, ICSM ’98, pages 24– 31. IEEE, 1998.
25. S.A. Ajila and R.T. Dumitrescu. Experimental use of code delta, code churn, and rate of change to understand software product line evolution. *Journal of Systems and Software*, 80(1):74 – 91, 2007.
26. Ostrand, T. J., Weyuker, E.J, Bell, R.M., "Where the Bugs Are," Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2004.pp 86-96.
27. A. Gupta, O. P. N. Slyngstad, R. Conradi, P. Mohagheghi, H. Ronneberg, E. Landre, “A Case Study of Defect-Density and Change-Density and their Progress over Time,” in 11th European Conference on Software Maintenance and Reengineering, 2007. CSMR ’07, Mar. 2007, pp. 7–16.
28. S. M. A. Shah, M. Morisio, M. Torchiano, “An Overview of Software Defect Density: A Scoping Study,” in Software Engineering Conference (APSEC), 2012 19th Asia-Pacific, vol. 1, Dec. 2012, pp. 406–415.
29. P. Klint. Estimate the post-release Defect Density based on the Test Level Quality. Universiteit Van Amsterdam.
30. E.W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(1- 4):251–266, 1986