Project Proposal for Software Measurement

SOEN-6611

Submitted to: Jinqiu Yang

Submitted by: Team G

Department of Computer Science and Software Engineering
Gina Cody School of Engineering and Computer Science
Concordia University

May 23, 2019

Contents

1 Team Information

Name	Student ID	Email
Maria Ahmed	40070844	ritu329@gmail.com
Shehnaz	40070137	shehnaz1191@gmail.com
Mahy Salama	40074737	mahyhanysalama@gmail.com
Gagandeep Jaswal	40066862	gagandeepjaswal227@gmail.com
Samaneh Shirdel Farimani	40075615	samaneh.shirdelk@gmail.com

2 Selected Metrics

Metric 1 : Test Coverage - Statement coverage
The statement coverage is also known as line coverage or segment coverage. The statement coverage covers only the true conditions. Through statement coverage we can identify the statements executed and where the code is not executed because of blockage.

Metric 2 : Test Coverage - Branch coverage
It covers both the true and false conditions unlikely the statement coverage. Branch coverage is a testing method, which aims to ensure that each one of the possible branch from each decision point is executed at least once and thereby ensuring that all reachable code is executed.

Metric 3 : Test Suit Effective - Mutation Score
Test suite is a collection of test cases that are used to test the software program. So, it is very important to have efficient test suite. Because effectiveness of testing is calculated by measuring the quality of the test suite. Mutation Score is a proxy measurement to measure the quality of the test suites means for test suite effectiveness. The main functionality in test suite is to find out real faults in a program. Mutation score measures the test suite ability to distinguish original program from many other small variations known as mutants. A test suite having high mutation score detects more real faults than the test suite having low mutation score[1].

Metric 4 : Complexity metric - McCabe complexity
Used to measure the complexity of the program. it measures the number of linearly independent paths that can be executed through a piece of source code. This metric can be used by high level managers to obtain the development status of the software project. It is beneficial to control and manage the entire development procedure.

Metric 5 : Software Maintenance Effort - Code Churn
Code Churn is a measure of the amount of the code changed within software over a period of time. It is found by checking the change history of a system which is recorded by version control systems. The version Control systems use a comparison technique to find number of lines added, changed or deleted by programmer by comparing the previous version of the system to new version. The results obtained are used to measure code churn.[2]
Code Churn is calculated by comparing the baseline version with new version and it is absolute value of code changed in system [3]. Nagappan and Ball have suggested seven measures to find code churn. The following two measures are selected from these seven measures-

$$\text{M}easure1 = \frac{\text{Churned LOC}}{\text{Total LOC}}$$

$$\text{M}easure2 = \frac{\text{Deleted LOC}}{\text{Total LOC}}$$

In Microsoft Azure, several different measures to calculate Code churn are suggested. From these measures, the one to calculate Total Churn is selected-

$$\text{Measure 3} = \text{Lines Added} + \text{Lines Deleted} + \text{Lines Modified [4]}$$

The terms used to define these measures are described below:-
Total LOC- number of lines of non-commented executable lines in the files comprising the new version.
Churned LOC – sum of added and changed lines of code between baseline and new version.
Deleted LOC- number of lines deleted between baseline version and new version.
To calculate the code churn for open source software's, various versions of software will be considered. We are planning to use above proposed three measures to find code churn in our open source software's for this project. The rational behind choosing this metric is described in Related Work section.

Metric 6: Software Quality Attribute - Defect density
Defect density is a measure of the total known defects divided by the size of the software entity being measured [5].

$$\mathrm{D}efectdensity = \frac{\text{number of known defects}}{\text{software size}}$$

A defect can be thought of as an occurred failure in the system. As we will collect data from open source software instead of defect density, we are using post release defect density. We will collect data for number of known defects using available issue tracker or bug tracker of the open source software.
Software size can be measured using line of code (LOC) or source line of code (SLOC). For our project we aim to use Thousands of source line of code (KSLOC). The rational behind choosing this metric is described in Related Work section.

$$\text{Post-release defect density} = \frac{\text{number of known bugs reported in issue tracker for a version}}{\text{KLOC of that version}}$$

3 Correlation Analysis

- Correlation (Metric 1 & 2) and Metric 3: Both statement and branch coverage of a test suite is performed to detect the faults. Studies is done for this by finding the relation between test suite size, coverage and effectiveness of programs. After performing the study on huge data like 31,000 test suites for five systems that consists of 724,000 lines of code, it is clear that there is a very low or moderate co-relation between coverage and effectiveness. Coverage is useful for just identifying the tested parts not for identifying the quality of test suite [1]. Some studies have shown values also that represents that there is moderate relation between coverage and test suite effectiveness even if we use real bugs. Like co-relation between coverage (statement and branch) and effectiveness of a test suite for HTTP Client is (i.e., 0.25 rpb 2 ¡ 0.49) and for Rhino, the relation is represented as (i.e., 0.49 rpb 2 ¡ 0.81)[7]. It is not highly co-related because sometimes if any segment of the code (either statement or whole branch) is not covered then it is difficult to find bugs in the code.

- Correlation between Metric (1 & 2) and Metric 4 For a Complete branch coverage (i.e., all independent paths are covered), McCabe complexity is used in determining the number of test cases that are necessary to achieve complete branch coverage where the number of test cases should be equal to the number of independent paths. From the research done in [6] the more complex the modules are, the more effort should be put on these modules for test and maintenance. Therefore, as the complexity number decreases (i.e. 1-10), the software is high testability and cost and effort are less. While, as the complexity number increases (i.e. ¿40), the software is not at all testable and cost and effort are very high.

- Correlation metric (1 & 2) and metric 6: Test coverage defines how much of the code are actually being exercised during a test. Test coverage can also reveal information about dead code or unreachable code. With increased test coverage we expect to find more defect so system with more test coverage are expected to have less post release defect density.

- Correlation between Metric (5 & 6) - Code Churn in software can occur because of several reasons. The source can be changed due to addition of new features, modifying existing functionality or deleting some code. Almost all the times, whenever code churn occurs. It leads to injection of defects and bugs in software.The rational is that high code churn will lead to high defect density.

4 Related Work

- Metric (1 & 2) -

  In ‘ Program State Coverage: A Test Coverage Metric Based on Executed Program States ‘ paper [1] they discussed that in software testing, different metrics are proposed to predict and compare test suites effectiveness and Mutation Score (MS) is one of most accurate metrics.
  They propose a novel test coverage metric, called Program State Coverage (PSC), which improves the accuracy of LC. PSC works almost the same as LC and it can be calculated by executing test suites only once. However, it further considers the number of distinct program states in which each line is executed. Their experiments on 120 test suites from four packages of Apache Commons Math and Apache Commons Lang showed that, compared to LC, PSC is more strongly correlated with normalized MS. As a result, they conclude that PSC is a promising test coverage metric.
  In ‘ Identifying Characteristics of Java Methods that May Influence Branch Coverage: An Exploratory Study on Open Source Projects ‘ paper[1] the main purpose of this paper is to present an investigation conducted by authors to identify the differences between methods whose branches were fully covered and the methods that have been partially covered. They conducted their investigation on 39 open source Java projects.

- Metric (1 & 2) and Metric 6 :

  Malaiya et all found out in their experiment that if branch coverage is 50 percent then if does not have any major affect on number of found bugs but if branch coverage is increased to 84 percent then that can have considerable affect on defect density[12].

- Metric 3 -

  The most related work regarding coverage and test suite effectiveness is done by Gopinath et al[8] and Inozemtseva et al [1]. They referred to a large number of projects from Git Hub and calculated the test cas0es coverage- that can be manually generated or automatically generated test cases and performed mutation analysis on it. The test cases that the author generated manually is collected from Git Hib and the automatic are produced by running Randoop (which is feedback- directed random testing approach). The author performed cross validation and he found that statement coverage is good for test suite effectiveness.
  Inozemtseva et al, experimented 5 open source systems and generated 31,000 test suites for these projects. When tested on the basis of statement and branch testing the expirements proved that code coverage is not related to test suite effectiveness.

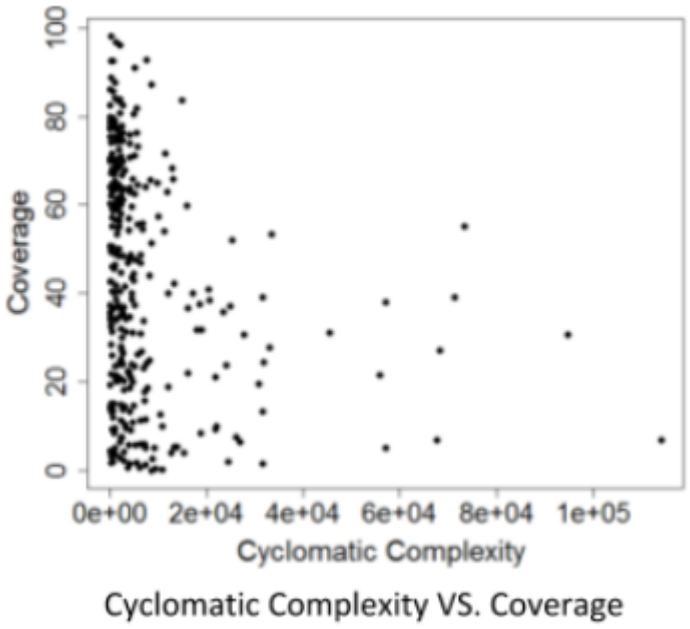
- Metric 4 -

  As per “An Analysis of the McCabe Cyclomatic Complexity Number”[13], in this study it first defines some measurement concepts like entity, attribute, process and result, then describes the McCabe Cyclomatic Complexity based on the control flow graph using the formula:

  $$v(G)^{independent\ cycle} = (e+1)^{edge+\ virtual\ edge} - n^{nodes} + p^{connectedco}$$

  Finally, it analyses this metric where the number of McCabe can be validated internally as an ordinal, ratio, and absolute scale. However, with the term ‘complexity’ being used to label the measure, users of the number derived from the above equation do not seem to interpret this number on a ratio scale, i.e. complexity of a program is twice another program if their numbers are doubled. For the empirical

study in [9], their study aims to understand and improve the state of testing in open-source software, and how to correlate code coverage information with other software metrics that can characterize the software development process, such as lines of code, cyclomatic complexity, and number of developers. They used "Sonar" an open-source platform to manage software Quality that combines various existing tools for getting code coverage, and executing unit test cases. Sonar can also compute the various software metrics such as lines of code, cyclomatic complexity, number of test cases etc. they come to conclusion that Cyclomatic complexity of software generally increases with an increase in the number of lines of code and that the coverage level decreases with an increase in the complexity of the code, as shown in the Figure below "The scatter plot between coverage and cyclomatic complexity".



Cyclomatic Complexity VS. Coverage

Cyclomatic complexity has a significant impact on testing. Thus, developers who are working on large and complex projects should put more emphasis on testing to improve the reliability of the software.

- Metric (5 & 6) -

In a study by "Munson et al, they observed that as a system is developed, the relative complexity of each changed module also changes. The rate of change in relative complexity serves as a good index of the rate of fault injection. They studied a 300KLOC(thousand lines of code) embedded real time systems with 3700 modules programmed in C. Code Churn metrics were found to be among most highly correlated with problem reports."[2]
In another study by "Ostrand et al, They used information such as file status as new, changed, unchanged files along with other 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."[2]These studies motivated us to consider Code Churn as Software maintenance effort metric for our project.

Measuring software quality can be a daunting task as there are lots of factor that affects the overall quality of a software. Daskalantonakis (1992) described Motorola's software metrics program in his paper "A Practical View of Software Measurement and Implementation Experiences Within Motorola". Motorola used Goal/Question/Metric paradigm and one of the goals was to "Decrease software defect density" and established metric was defect density. [10]
The environment in which an open source software is being developed is very different than the environment of a closed sourced software. Formal definition of software process or a defined organization culture is absent so maintaining software quality is necessary to achieve reliability. A way of achieving quality is to minimize number of faults(defects). According to Rahmani and Khazanchi defect density is a promising

metric to address quality issue in an open source software system. Rahmani and Khazanchi analysed forty-four randomly selected open source software projects retrieved from SourceForge.net and conclude that there is no statistically, significant linear relationship between KLOC and software defect density [11] According to these studies we considered post-release defect density as our software quality metric.

5 Selected Open Source Systems

1. Apache Accumulo
   - lines of code - 454K
   - Version number - Apache Accumulo 1.9.3
   - Version Control System - https://gitbox.apache.org/repos/asf/accumulo.git
   - Bug Tracking System - https://github.com/apache/accumulo/issues
2. Apache Brooklyn
   - lines of code - 797K
   - Version number - Apache Brooklyn 0.11.0
   - Version Control System - https://git-wip-us.apache.org/repos/asf/brooklyn.git
   - Bug Tracking System - https://issues.apache.org/jira/browse/BROOKLYN/
3. Apache CloudStack
   - lines of code - 1.35M
   - Version number - Apache CloudStack 4.3.2
   - Version Control System - https://git-wip-us.apache.org/repos/asf/cloudstack.git
   - Bug Tracking System - https://issues.apache.org/jira/browse/CLOUDSTACK
4. Apache Isis
   - lines of code - 292K
   - Version number - Apache Isis 1.17.0
   - Version Control System - https://git-wip-us.apache.org/repos/asf/isis.git
   - Bug Tracking System - https://issues.apache.org/jira/browse/ISIS
5. Apache Ivy
   - lines of code - 305K
   - Version number - Apache Ivy 2.4.0
   - Version Control System - https://gitbox.apache.org/repos/asf/ant-ivy.git
   - Bug Tracking System - https://issues.apache.org/jira/browse/IVY

6 Resource Planning

Name	Responsibility
Maria Ahmed	Data Collection for Metric 6 and will prove correlation between Metric 1 & 2 and 6 and Metric 5 & 6
Shehnaz	Data Collection for Metric 5 and will prove correlation between Metric 1 & 2 and 6 and Metric 5 & 6
Mahy Salama	Data Collection for Metric 4 and will prove correlation between Metric 1 & 2 and 4
Gagandeep Jaswal	Data Collection for Metric 3 and will prove correlation between Metric 1 & 2 and 3
Samaneh Shirdel	Data Collection for Metric 1 and 2

7 References

1. Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, New York, NY, USA, 435-445. DOI: https://doi.org/10.1145/2568225.2568271

2. N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.

3. 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, vol. 80, no. 1, pp. 74–91, 2007.

4. KathrynEE, "Analyze and report on code churn and code coverage - TFS," Analyze and report on code churn and code coverage - TFS — Microsoft Docs. [Online]. Available: https://docs.microsoft.com/en-us/azure/devops/report/sql-reports/perspective-code-analyze-report-code-churn-coverage?view=azure-devops-2019.

5. L. Westfall, "Defect Density," Available: http://www.westfallteam.com/Papers/defect_density.pdf.

6. Honglei, T., Wei, S., Yanan, Z. (2009, December). The research on software metrics and software complexity metrics. In 2009 International Forum on Computer Science-Technology and Applications (Vol. 1, pp. 131-136). IEEE.

7. "Code Coverage and Test Suite Effectiveness: Empirical ..." [Online]. Available: http://www.mysmu.edu/faculty/davidl coverage.pdf.

8. Rahul Gopinath, Carlos Jensen, and Groce Alex. Code coverage for suite evaluation by developers. In ICSE, pages 72–82, 2014

9. Kochhar, P. S., Thung, F., Lo, D., Lawall, J. (2014, December). An empirical study on the adequacy of testing in open source projects. In 2014 21st Asia-Pacific Software Engineering Conference (Vol. 1, pp. 215-222). IEEE.

10. M. Daskalantonakis, "Correction to A practical view of software measurement and implementation experiences within Motorola," IEEE Transactions on Software Engineering, vol. 19, no. 2, pp. 199–200, 1993

11. C. Rahmani and D. Khazanchi, "A Study on Defect Density of Open Source Software," 2010 IEEE/ACIS 9th International Conference on Computer and Information Science, Yamagata, 2010, pp. 679-683

12. Y. K. Malaiya, M. N. Li, J. M. Bieman and R. Karcich, "Software reliability growth with test coverage," in IEEE Transactions on Reliability, vol. 51, no. 4, pp. 420-426, Dec. 2002.

13. Abran, A., Lopez, M., Habra, N. (2004). An analysis of the McCabe Cyclomatic complexity number. In Proceedings of the 14th International Workshop on Software Measurement (IWSM) IWSM-Metrikon (pp. 391-405)