SOFTWARE MEASUREMENT

SOEN - 6611

TEAM - E

Replication Package

https://github.com/VigneswarM/SoftwareMeasurement

Team Details

Name	Student ID	Email
Kasyap Vedantam	40042347	vkhashyap@gmail.com
Sai Krishna Alla	40069746	saikrishnaalla3@gmail.com
Alekya Karicherla	40059347	alekhya.karicharla9@gmail.com
Vigneswar Mourouguessin	40057918	imvigneswar@gmail.com
Ezhilmani Aakaash	40071067	aakaash07@gmail.com

Measuring Software Quality with Correlation of Different Software Metrics

Kasyap Vedantam (40042347) Sai Krishna Alla (40069746) Alekya Karicherla (40059347)
Vigneswar Mourouguessin (40057918) Ezhilmani Aakaash (40057918)

Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada

Abstract:
The ability to deliver a software on time, within budget and with expected functionality is critical to all software customers. This can be achieved by predicting, controlling, understanding and evaluating the software quality on regular basis. To perform above all one should be able to measure both the external and internal attributes. The external attributes include properties that are measured with user experiences such as reliability, efficiency, usability etc. Whereas the internal attributes cover the properties visible only to developer team such as size, complexity, coding standards etc.

Our goal of this study is to compute six software metrics on five open source projects and analyze the correlation between them. The metrics considered in this study are Coverage Metrics like Statement coverage & Branch Coverage, Test suite Effectiveness, Complexity metric, Software maintenance metrics and quality metrics.

I. Introduction

The main objective of this project is to analyze five open source projects, collect the metrics and conduct the correlation analyses. Each correlation has a rationale which is either approved or denied based on the tests conducted on each project. We considered five open source JAVA projects each having at least 100K SLOC with an issue-tracking system.

The following are the metrics considered for correlation and their rationale.

Metric 1	**Metric 2**	**Rationale**
Code Coverage (Statement & Branch Coverage) Metric	Test suite effectiveness	Test suites with higher coverage might show better test suite effectiveness.
Complexity Metric	Code Coverage Metric	Classes with higher complexity are less likely to have high coverage test suites.
Coverage Metric	Software Quality (Post release defect density)	Classes with low test coverage contain more bugs
Software Maintenance (Code Churn)	Software Quality (Post Release Defect Density)	The correlation between code churn and defect density is proportional because of the effects on one another.

II. Methodology

We selected five JAVA open source projects for correlation analysis. All the projects were selected based on a selection criterion of having Issue Tracking System, multiple versions and at least 100K LOC.

All the finally selected projects were Apache Commons projects as other projects were creating issues like requiring different versions of jdk, build issues in different development platforms etc. And, all the Apache Commons projects satisfy the selection criterion required for conducting our tests to find the correlation.

The following table summarizes the details about the projects that are used in this study.

Project Name	Description	SLOC	Issue Tracking System
Apache Commons-Configurations	Provides a generic configuration interface which enables a Java application to read configuration data from a variety of sources.	91K	JIRA
Apache Commons-Codec	Provides implementations of common encoders and decoders.	47K	JIRA
Apache Commons-Collections	The package contains types that extend and augment the Java Collections Framework.	127K	JIRA
Apache Commons-Math	Commons Math is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language or Commons Lang.	486K	JIRA
Apache Commons-Pool	The Apache Commons Pool open source software library provides an object-pooling API and several object pool implementations.	28K	JIRA

III. Metrics

1)Code Coverage (Metric 1 & 2):

Code coverage is used to determine the percentage of code covered by the test cases. This can be used as a measurement to determine which statements or branches of code has been executed by the test cases and vice-versa.

Statement Coverage (Metric 1) is a white box technique used to cover all the possible lines, paths and statements in the code.

*Statement Coverage = (No. of Executed Statements/Total No. of Statements) *100*

Branch Coverage (Metric 2) is used to calculate the number of possible branches executed during the test run.

*Branch Coverage = (No. of Executed Branches/Total No. of Branches) * 100*

2) Mutation Score (Metric 3):

Mutation testing is used to identify defects in the code and to assess the quality of the test cases by introducing mutants. Mutants are created by changing an operand or operator in certain statements. Now by test run, the test cases should be able to identify the mutants and kill them.

*Mutation Score = (No. of Killed Mutants/Total No. of Mutants) * 100*

Test cases are mutation adequate if the score is 100%.

3) McCabe Complexity or Cyclomatic Complexity (Metric 4):

This can be used to identify the complexity of the code by measuring the number of linearly independent paths in the source code of a software application. Independent paths are the paths that contains at least one edge which has not been traversed in other paths. Cyclomatic complexity is often useful in knowing the number of test cases that might be required for independent path testing Less complexity implies well written code, high testability.

*McCabe Complexity = No. of Edges - No. of Vertices + 2 * Connected components;*

McCabe Complexity = No. of Control Predicates + 1;

McCabe Complexity = Missed complexity/ Total Complexity

4) Code Churn (Metric 5):

Code churn is a measure of the amount of code change 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. Relative Code

churn is used as some studies show that absolute code churn is a poor indicator of post-release defect density.

Relative code churn= Churned LOC / Total LOC; Churned LOC = added LOC + modified LOC

5) Post Release Defect Density (Metric 6):

Defect Density is the number of defects confirmed in software/module during a specific period of operation or development divided by the size of the software/module. It enables one to decide if a piece of software is ready to be released. Defect density is counted per thousand lines of code also known as KLOC.

Defect Density = Defect Count/size of the release

Post-Release Defect Density

$DD_{Post-release} = |D_{Post-release}| / KSLOC$

Where $|D_{Post-release}|$: is the total number of post-release defects,

KSLOC: are the total number of added or modified SLOC, where SLOC are the total number of 1000 source lines of code.

IV. Data Collection and Analysis

Statement Coverage, Branch Coverage & McCabe Complexity:

Statement and Branch Coverage can be measured by using JACOCO. Each project is imported in IntelliJ IDE. pom.xml of each project will be updated with Jacoco plugin, if required. Running the project under "Coverage as" runs the test cases and gives a coverage report. Detailed coverage report is exported as HTML and CSV files.

We used the open source tool Jacoco to measure two types of coverage: statement and decision coverage and McCabe Complexity. It requires to add dependency in pom.xml as below.

```
<plugin>
        <groupId> org.jacoco </groupId>
   <artifactId> jacoco-maven-plugin </artifactId>
        <version> 0.7.7.201606060606</version>
   <executions>
      <execution>
         <goals>
            <goal> prepare-agent </goal>
         </goals>
      </execution>
         <execution>
         <id> report </id>
         <phase> prepare-package </phase>
            <goals>
               <goal> report </goal>
            </goals>
         </execution>
   </executions>
</plugin>
```

And then execute the following command to generate html and csv reports.

mvn clean install

The generated html and csv reports can be found in directory target/site.

Mutation Score:

PiTest is used to measure the mutation score by seeding mutants in the application code and finding the validity and effectiveness of test suite.

Pom.xml is updated with pitest plugin, which can give us the mutation score.

We used the open source tool PIT to generate faulty versions of our programs. In addition to that, here we run mutation testing on entire project than some classes.

Following is the dependency plugin added to pom.xml

```
<plugin>
        <groupId> org.pitest </groupId>
   <artifactId> pitest-maven </artifactId>
        <version> 1.4.2 </version>
   <configuration>
      <outputFormats>
         <param> HTML </param>
         <param> CSV </param>
      </outputFormats>
         </configuration>
</plugin>
```

And then, the following commands are executed to run PIT testing on entire project.

mvn clean install

mvn org.pitest:pitest-maven:mutationCoverage

Report will be generated at target/pit-reports/*.

Relative Code Churn:

We used CLOC tool to calculate the number of lines which are added and modified to current version with respect to recent previous version. CLOC is known to run on different operating systems. It can be installed via package managers depending on the type of Operating System. It is platform independent tool and uses very famous Diff Algorithm.

CLOC is command line tool which can be used to measure lines of code as well as different between two files or directories. It takes the input as a file or directory and outputs SLOC, number of modified lines between versions, number of commented lines, number of blank lines etc.

Two versions of project need to be considered along with all the commits in between the versions.

We used CLOC to count the lines of code (added and modified lines) and performed the below calculations sequentially to obtain results.

- Firstly, execute *command cloc-1.64.exe proj_V1*, this will provide loc of V1(Version 1)
- Perform the same action for the next commit (if any) For e.g.*: cloc-1.64.exe proj_V1_Commit1*
- Now execute *command cloc-1.64.exe -diff proj_V1 proj_V1_Commit1 Output*
- Calculate LOC (Modified + Added) for all commits between versions and for version2 as well.
- *Relative Churned Code = Summate all calculated LOC's (Modified + Added)/ version2 LOC*

Post Release Defect Density:

To calculate this, defects after project release needs to be counted from issue tracker. In our case, JIRA is the issue tracker.

First, we search JIRA issue tracking system for project (Project main page --> Issue tracking --> All Apache Bugs). Proper and latest issue tracker link for the project can be found on project site.

Then by using advanced search filter, we find number of bugs, and record it for 5 different version of each project.

- Click on Advanced link, enter query *project =xxx and affectedVersion = xx ORDER BY key desc.*
- Take count of bugs.
- Defect Density = *No. of Bugs/ Size of release.*

Size of release = LOC --> Calculated using cloc

Tools Used

Statement & Branch Coverage: JACOCO

Mutation Score: PiTest

McCabe Complexity: JACOCO

Relative Code Churn: CLOC

Post Release Defect Density: CLOC & Jira

V. Results

A. Coverage Metrics & Test-suite Effectiveness

Test effectiveness is the measure of how well the given test classes are at finding the errors in the code being tested. We can assume that if the class is having higher coverage, then the mutation score will automatically be higher provided the test suites are proper and well-defined.

The Running time for all the projects besides Configuration and Math were around ~40mins while Configurations took 1hr and Math around 5hrs. The correlation between these were done using an online Jupyter Notebook (Python). The required .csv file was formatted and supplied to it for her required analysis.

The following are the obtained results.

Statement Coverage vs Mutation Score

Project Name	V1
Commons- Configuration	0.2125
Commons- Codec	0.3565
Commons - Collections	0.1846
Commons - Math	0.3418
Commons - Pool	0.457

Table 1: Statement Coverage Vs Mutation Score

Branch Coverage vs Mutation Score

Project Name	V1
Commons- Configuration	0.5
Commons- Codec	0.21325
Commons - Collections	0.06
Commons - Math	0.1231
Commons - Pool	0.168

Table 2:Branch Coverage Vs Mutation Score

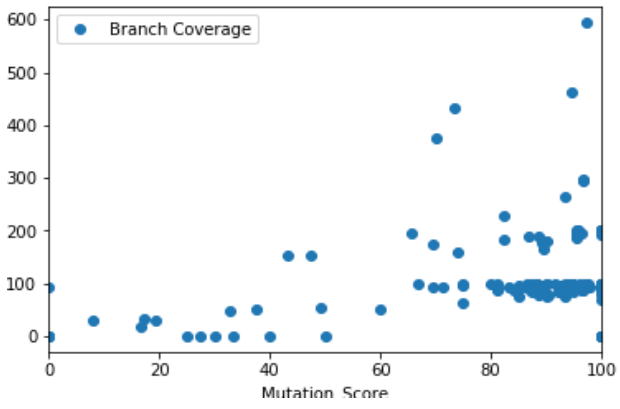

Fig. 1. Correlation between Branch Coverage and Mutation Score

Average Values for Table 1 = ~0.3

Average Values for Table 2 = ~0.2

Statement Coverage has **moderate positive correlation** with Mutation Score.

Branch Coverage has **moderate positive correlation** with Mutation Score.

B. Complexity & Code Coverage Metrics

Code coverage refers to how much part of the code is visited/covered when he program is run and tested. This is a metric is valued high due it to ability to find areas of used code or complexity.

It is usually assumed that if the lines of code are increased, the amount of time required to compute these standards is high. This can also mean that if a routine is of higher complexity. Then, computation of this metric becomes quite difficult.

We have performed the correlation of these metrics on the below mentioned projects and for our better understanding, we have opted for 3 version for each of the projects.

Statement Coverage vs McCabe Complexity

Project Name	V1	V2	V3
Commons-Configuration	-0.451	-0.4190	-0.409
Commons- Codec			
Commons - Collections	-0.39	-0.397	-0.397
Commons - Math	-0.336	-0.336	-0.3362
Commons - Pool	-0.280	-0.272	-0.274

Table 3: Statement Coverage Vs McCabe Complexity

From the table above, most of them are well under ~-0.3 besides **Configuration** project making it evident that is **small negative correlation**.

Branch Coverage vs McCabe Complexity

Project Name	V1	V2	V3
Commons-Configuration	-0.354	-0.3177	-0.318
Commons- Codec	-0.344	-0.278	-0.3581
Commons - Collections	-0.26	-0.262	-0.260
Commons - Math	-0.125	-0.1255	-0.1257
Commons - Pool	-0.2095	-0.182	-0.167

Table 4: Branch Coverage Vs McCabe Complexity

From the table above, most of them are well under ~-0.3 besides **Configuration** project making it evident that is **small negative correlation**.

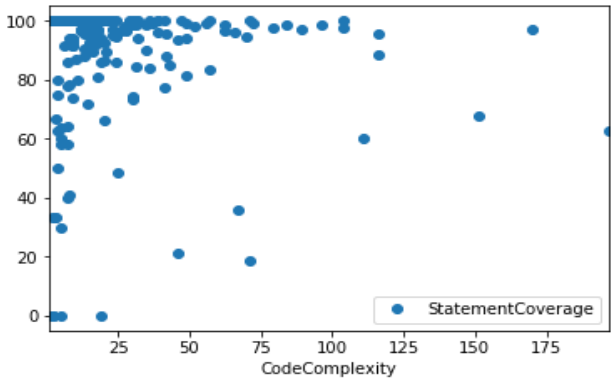

Fig. 2. Correlation between Statement Coverage and McCabe Complexity

C. Coverage & Software Quality

The Coverage is here is based on the visited part of the code.

For Statement, it is the amount of statements visited divided by the total number of statements in the code.

For Branch, it is the number of branches visited divided by the total number of branches.

Project Name	Statement vs Defect D	Branch vs Defect D
Commons-Configuration	-0.5	0.499
Commons- Codec	0.9	1.0
Commons - Collections	-1.0	-1.0
Commons - Math	-0.359	0.0512
Commons - Pool	-0.8660	-0.8660

Table 5: Correlation between Branch Coverage and Statement Coverage and Defect density

Based on the observed table from table V, the can draw the following conclusion.

1. Statement vs DD shows that besides Math and Codec, rest of the projects have **high negative correlation**.

2. Branch vs DD shows that besides Math and Codec, rest of the projects have **moderate negative correlation**.

Also, there are a few erratic values such as for Codec. Hence, we have ignored it for the above conclusion.

D. Software Quality & Software Maintenance

Defect Density is calculated based on the number of bugs that were calculated using the JIRA tool. This was done

at the project level for all 5 projects (Single and Final Versions).

Project Name	Spearman Ranking
Commons- Configuration	-0.5
Commons- Codec	-0.5
Commons - Collections	1.0
Commons - Math	-0.1
Commons - Pool	-0.5

Table 6: Spearman Ranking

The values seem to range between positive and mostly negative. (Collections).

The values also are between the -1.0 to +1.0 mark for the spearman ranking coefficient.

As per our hypothesis, it is evident from the tabulated values that the Defect Density Metric and Code Churn Metric **do not have much in common. (or correlated).**

VI. Related Work

A. Coverage Metrics & Test Suite Effectiveness:

The Influence of Size and Coverage on Test Suite Effectiveness and Coverage Is Not Strongly Correlated with Test Suite Effectiveness

The papers mentioned above concluded that there exists no linear relationship between size, coverage and effectiveness but a nonlinear relationship does exist. The same has been supported by two other papers [1][2]. All these papers states that coverage is correlated with test suite effectiveness when size is controlled. There exists a moderate to high correlation when the suite size is ignored and low to moderate when size is controlled. These suggest that coverage is not alone a good predictor of test suite effectiveness.

B. Complexity & Code Coverage Metrics:

Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric

The papers mentioned above concluded that there exists a linear relationship between cyclomatic complexity and code coverage. Cyclomatic complexity achieves higher code coverage by uncovering maximum number of bugs with minimum number of test cases. Path coverage helps achieving 100% code coverage than branch and statement coverage. Cyclomatic complexity indicates complexity based on independent paths. Hence, we conclude that complexity is strongly correlated with code coverage.

C. Coverage & Software Quality:

Estimating Defect Density Using Test Coverage

The results show that the increase in software quality is accompanied by increase in at least one the coverage measure. It was also observed that decrease in reliability is accompanied by a decrease in at least one code coverage measure. Hence, we conclude that there is a strong correlation between coverage and software quality.

D. Software Maintenance & Software Quality:

Use of Relative Code Churn Measures to Predict System Defect Density

The paper states that relative measures of code churn can be used as an excellent predictor for system defect density. A case study is performed on Windows Server 2003 using 8 relative code churn measures. The correlation is performed using Spearman rank Correlation. This paper concludes that increase in relative code churn measures results in an increase in system defect density, and relative code churn measures can be used as effective predictors of system defect density and to discriminate between fault and non-fault binaries.

VII. Conclusion

We were able to compute and correlate all the metrics with one another allowing us to come to few assumptions and conclusions:

1. Moderate positive correlation for Coverage vs Mutation Score.
2. Small negative correlation for Coverage vs McCabe Complexity.
3. Moderate to High negative correlation for Coverage vs Defect Density.
4. No correlation between Defect density and Code Churn. This statement was concluded since the values seemed to fluctuate when correlated, between positive to negative. This may attribute to the fact that the definitions of Code Churn may defer from project to project. (Addition + Modified) can have adverse results for chosen open-source projects.

VIII. References

[1] PAVNEET SINGH KOCHHAR, FERDIAN THUNG, and David LO. Code Coverage and Test Suite Effectiveness: Empirical Study with Real Bugs in Large Systems. (2015). 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER). 560-564. Research Collection School Of Information Systems. Available at:

https://ink.library.smu.edu.sg/sis_research/2974

[2] https://www.theserverside.com/feature/How-to-calculate-McCabe-cyclomatic-complexity-in-Java

[3] Eclemma.org. (2019). JaCoCo - Coverage Counter. Available at: https://www.eclemma.org/jacoco/trunk/doc/counters.html. Accessed on: 7 February 2019

[4] Watson, A. & McCabe, T., (NIST 500-235), Structured Testing: a testing methodology using the cyclomatic complexity metric, Special Publication 500-235, National Institute of Standards and Technology, Sept. 1996

[5] Shepperd, Martin. (1988). A Critique of Cyclomatic Complexity as a Software Metric. Software Engineering Journal.

[6] Calculating Mccabe's Cyclomatic Complexity Metric And Its Effect On The Quality Aspects Of Software, International Journal of Innovative Research and Creative Technology (www.ijirct.org), ISSN:2454-5988, Volume-3 Issue-5 page no.10-22, March-2018,

[7] A Large-Scale Study of Test Coverage Evolution Michael Hilton Carnegie Mellon University, Jonathan Bell George Mason University , Darko Marinov University of Illinois

[8] Examining the Effectiveness of Testing Coverage Tools: An Empirical Study - Khalid Alemerien & Kenneth Mage Computer Science Department, North Dakota State University Fargo, North Dakota, USA

[9] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In Proc. of the Int'l Symposium on Software Testing and Analysis, 2009.

[10] 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

[11] Estimating Defect Density Using Test Coverage Yashwant. K. Malaiya and Jason Denton: https://www.researchgate.net/publication/2460146_Estimating_Defect_Density_Using_Test_Coverage

[12] Marek Leszak, Dewayne E. Perry and Dieter Stoll, "A Case Study in Root Cause Defect Analysis," in Proceedings of the 22nd international conference on Software engineering, June 2000, pp. 428-437.

[13] T.W.Williams, M.R.Mercer, J.P.Mucha, and R.Kapur, "Code Coverage, What Does It Mean in Terms of Quality?" in 2001 IEEE Proceedings Annual Reliability and Maintainability Symposium, 22-25 January 2001, pp. 420-424 : https://digital.library.ryerson.ca/islandora/object/RULA%3A2214/datastream/OBJ/view

[14] Sebastian G. Elbaum and John C. Munson, "Code Churn: A Measure for Estimating the Impact of Code Change", Computer Science Department University of Idaho Moscow, ID 83844-1010 https://pdfs.semanticscholar.org/c31c/a6c0230b4d902e1fa8741fa82c171a149955.pdf

[15] Nachiappan Nagappan (Department of Computer Science North Carolina State University) and Thomas Ball (Microsoft Research), "Use of Relative Code Churn Measures to Predict System Defect Density"

[16] Spearman Correlation Coefficient: https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient.

[17] https://www.guru99.com

[18] http://www.bullseye.com/coverage.html

[19] https://pjcj.net/testing_and_code_coverage/paper.html