SOFTWARE MEASUREMENT

SOEN - 6611

TEAM - E

Replication Package

//githublink

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.We considered five open source JAVA software 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 Coverage & 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 and at least 100K LOC.

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	JITA
Apache Commons-Collections	The package contains types that extend and augment the Java Collections Framework [1].	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

Selected Open-Source Systems

We selected 5 open source java projects from variety of application domain. The Apache common collection is Framework was a major addition in JDK 1.2. It added many powerful data structures that accelerate development of most significant Java applications. Since that time, it has become the recognized standard for collection handling in Java.

The Commons Configuration software library provides a generic configuration interface which enables a Java application to read configuration data from a variety of sources. Commons Configuration provides typed access to single, and multi-valued configuration parameter. Commons IO is a library of utilities to assist with developing IO functionality. The standard Java libraries fail to provide enough methods for manipulation of its core classes. Apache Commons Lang provides these extra methods. 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.

III. Metrics

1)Code Coverage:

Software testing is utilized to test various functionalities of a program or framework and to guarantee that given a lot of data sources the framework creates the normal outcomes. A test sufficiency standard characterizes the properties that must be fulfilled for a careful test. Code coverage, which estimates the percentage of code executed by test cases, is frequently utilized as an intermediary for test sufficiency. The percentage of code executed by test cases can be estimated by different criteria, including the percentage of executed source code lines (Statement coverage - Metric 1), and the percentage of executed branches (branch coverage Metric 2).

2)Mutation Score (Metric 3):

A mutant is another adaptation of a program that is created by making a small syntactic change to the original program. For instance, a mutant could be created by changing a constant, negating a branch condition, or removing a method call. The resultant mutant may create a similar output as the original program, wherein case it is called an equivalent mutant. For instance, if the equality test in the code snippet were changed to if (index >= 10), the new program would be an equivalent mutant.

int index = 0;

```
while (true) {
        index++;
        if (index == 10) {
        break;
        }
}
```

Mutation testing tools, for example, PIT produce numerous mutants and run the program's test suite on each one. If the test suite fails when it is run on a given mutant, we say that the suite kills that mutant. A test suite's mutant coverage is then the fraction of non-equivalent mutants that it kills. Equivalent mutants are excluded because they cannot, by definition, be detected by a unit test.

If a mutant is not killed by a test suite, manual inspection is required to determine if it is equivalent or if it was simply missed by the suite. This is a time-consuming and error-prone process, so studies that compare subsets of a test suite to the master suite often use a different approach: they assume that any mutant that cannot be detected by the master suite is equivalent. While this technique tends to overestimate the number of equivalent mutants, it is commonly applied because it allows the study of much larger programs.

The mutation score is defined as the percentage of killed mutants with the total number of mutants. Test cases are mutation adequate if the score is 100%. Test cases are mutation adequate if the score is 100%.

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

Cyclomatic complexity measures the number of linearly independent paths in the source code of a software application. This measure increases by 1 whenever a new method is called or when a new decision point is encountered, such as an if, while, for, &&, case, etc. Cyclomatic complexity is often useful in knowing the number of test cases that might be required for independent path testing and a file or project with low complexity is usually easier to comprehend and test.

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. 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.

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 know as KLOC.

Defect density is computed using historical data of the subject projects from the respective issue tracking systems. Our open source subject projects are available on GitHub and having issue tracking system as JIRA, which allows us to process the status and the history of issue reports. Defect density is not directly dependent on failing tests, but on problems found and reported by project collaborators and users. Although we did not verify each reported defect individually, they are usually reported for the released versions of the systems, meaning that the defects were not found during testing. Hence, we assume that higher defect density rate is an indicator of unsuccessful testing and lower quality of the test suite's the respective issue tracking systems.

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. Adding Jacoc in Project and Runnig was a simple process. 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> prepare-agent </goal>
        </goals>
      </execution>
  </executions>
</plugin>
```

and executing it via following command, will generate HTML and CSV reports in target/site folder.

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 one more pitest plugin,which can retrieve the mutation score.The following command is used to generate the mutation report

mvn org.pitest:pitest-maven:mutationCoverage.

We used the open source tool PIT to generate faulty versions of our programs. Using PIT was like Jacoco, required to add dependency in pom.xml. In addition to that, here we run mutation testing on entire project than some classes.

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

We run this command to run PIT testing on entire project.

mvn org.pitest:pitest-maven:mutationCoverage -X

To calculate mutation score for each class from csv, we use below formula. PIT in its HTML reports also counts as per below formula.

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 can be installed through number of package managers such as npm(node package manager), choco(windows package manager), yum or apt(linux package manager) and many more. It is platform independent tool and uses very famous Diff Algorithm. There isn't one true diff algorithm, but several with different characteristics. The basic idea is to find a 'modification script' that will turn Text A into Text B. They use modification operations such as insertion and deletion. Usually, you'd want the minimal number of changes required, but some application also must weigh operations depending on how much text each one affects, or other factors.

CLOC is command line tool which can be used to measure lines of code as well as different between two files or directories. We executed below command which showed us output as per below screenshot. We considered only important change in lines of code such as Java, XML and ignoring HTML.

Cloc –diff file1.zip file2.zip

CLOC Line Difference

CLOC can also be used to measure SLOC for current version which works as a relative measure to this metric.

Post Release Defect Density:

First, we search JIRA issue tracking system for project and allow public access to all the issues filed in the tracking system. Proper and latest issue tracker link for the project can be found on project site. Then to count number of bugs raised for project, we apply following filter in its advance search.

For each bug, JIRA records the affected version of the software. We collected all the closed and resolved bugs for the checked-out version of the software. We perform this step manually for each software project, as each project has a unique name used by JIRA and each project has a different checked out version. We obtained the JIRA name of each project by searching the project's website. For example, the project commons-collections in our dataset, for which we use version 4.2, has JIRA name COLLECTIONS.

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

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

B. Complexity & Code Coverage Metrics

C. Coverage & Software Quality

D. Software Quality & Software Maintenance

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

C. Coverage & Software Quality:

Estimating Defect Density Using Test Coverage

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

VIII. References