
# Project Proposal
# SOEN 6611 – Software Measurement

16th February 2020

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

## Team E



| Name | Student ID | Contact Information |
|---|---|---|
| Hetvi Shah | 40089272 | hetvishah171995@gmail.com |
| Khyatibahen Pragajibhai Chaudhary | 40071098 | khyati.chaudhary1996@gmail.com |
| Sarvesh Hiten Vora | 40081458 | vorasarvesh99@gmail.com |
| Satish Chanda | 40091187 | sathishchandas@gmail.com |
| Venkat Mani Deep Chandana | 40080924 | venkatmanideep553@gmail.com |

1. Selected Metrics and Correlation Analysis

   a. **Metric 1&2: Statement Coverage and Branch Coverage**

   Statement coverage and Branch coverage are test coverage metrics. The outcome of these metrics is various paths that are useful for defining unit test cases. It's a difficult task for the testing team to cover all modules with test cases. Therefore, one can be sure about the complete coverage (100% coverage) of the project and can be well documented.

   Statement coverage is a white box testing approach. It is also known as Line coverage. It is used to measure and calculate the number of executable statements in a program. It is also used to check the quality of a program. Every code block is executed in statement coverage.

$$\text{Statement coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}}$$

   Branch coverage is also known as decision coverage. It covers all the edges and measures every branch in the program and fraction of independent code segments. It also helps us to find out which sections of code don't have any branches.

$$\text{Branch coverage} = \frac{\text{Number of executed branches}}{\text{total number of branches}}$$

   **Hypothesis:** High branch and statement coverage converges to easy and atomic tests for the test suits.

   b. **Metric 3: Mutation Score**

   Mutation testing is a method of testing where we change certain statements in the source code and check if the test cases can find errors. It is a type of White Box Testing which is mainly used for Unit Testing.
   The importance of mutation testing is, it helps to understand how well our code could handle the faults in the program, Mutation is nothing but a small change in the statement of the code.
   Process of mutation testing with an example:

   i. Mutants (Faults) are introduced into the code and the goal of the mutants is to fail the effectiveness of the test cases.
   ii. Test cases are introduced to the mutant program and the original program.
   iii. Compare the results of the mutant program as well as the original program.
   iv. If the code was able to identify the mutant and produce different results compared with the original program, this means that the mutant is identified and killed.
   v. If the test cases are unable to identify the mutant and produce the same results as in the original code, more effective test cases with refactored code is used to kill the mutants.

**Example of a mutant**: if ( x > y ) to if ( x < y ).

Mutation testing is usually very time consuming, so we generally use a set of automation tools like Stryker, PIT Testing, etc.

$$\text{Mutation Score} = \frac{\text{Killed Mutants}}{\text{Total number of Mutants}} * 100$$

$$\text{Total number of mutants} = \frac{\text{Number of mutants}}{\text{faults introduced into the code}}$$

Killed Mutants = Number of mutants that are Identified (by producing different results) by the code.

**Hypothesis:** The higher the mutation score, better the test suits are. Low mutation score will lead to improved test suits for complete coverage. If the test case doesn't find any faults, then the test case is not correctly coded and thus the mutation score=0%. But if all the faults are recognized then according to the formula of mutation score the value of mutation score=100%.

c. **Metric 4: Cyclomatic Complexity (McCabe)**

Cyclomatic complexity is a software metric used to indicate the complexity of the given software by calculating the independent paths through your source code structure. Path testing helps us to cover all the statements at least once in the program testing period. It checks each linearly independent path which mean the total number of test cases is equal to cyclomatic complexity of the program.

Step followed:

i. Construction of a control flow graph.
ii. Calculation of independent paths.
iii. Cyclomatic complexity calculation.
iv. Design of test cases according to the statement and branch coverage of the CFG.

$$McCabe(CC) = E - N + 2P$$
$$McCabe(CC) = D + 1$$

E = The number of edges of the graph.
N = The number of nodes of the graph.
P = The number of connected components.
D = The number of control predicate.

McCabe Complexity Metric will be calculated by a plugin of eclipse called Jacoco.

**Hypothesis:** The higher the cyclomatic complexity, more difficult it is to maintain, test and refactor. McCabe proposed a way in which we can determine the complexity of a method, which basically counts one for each place whenever the flow changes from a linear flow.

d. **Metric 5: Software Science Effort**

Software Science Effort (E): An estimation of programming effort based on the number of operators and operands. It is a combination of other Software Science metrics.

To calculate the Software Science Effort, we need various parameters.
Number of distinct operators(D) like if, while, <, <=, etc.
Number of distinct operands(E) like variables, constants, etc.
Total occurrence of operators(F).
Total occurrence of operands(G).
Vocabulary, $K = D + E$
Observed length, $L = F + G$
The complete volume, $L.\log_2 K$
The Difficulty, $D' = \frac{D.G}{2E}$
Software Science Effort, $E = V.D'$

Algorithm: Scan the whole module code to compute parameters and perform calculations.

**Hypothesis:** Maintainability should decrease as the effort increases.

e. **Metric 6: Post-release defect density**

The Post-release defect density is defined as the number of defects identified in software during a specific period of operation or development divided by the size of the software. It is often used as a related measure of quality. KLOC is the defect density per 1000 lines of code.

$$\text{Defect Density} = \frac{\text{Number Of Defects Confirmed}}{\text{Size of Software}}$$

Defect Density and Software Quality is inversely proportional. That is if the Defect Density increases the Software Quality decreases and vice versa. To calculate Post-release defect density, we will use the issue tracker of the respective project and will calculate it using the above formula.

**Hypothesis:** Higher post-release defect density is equal to extreme refactoring and regression testing.

f. Correlation Analysis

Between Metric 1 and Metric 3: If we consider high mutation score then it led to high statement coverage. Changing some code and still getting high statement coverage converges to high testability.

Between Metric 2 and Metric 3: By using mutation technique we can generate various versions of the same code with minor changes that are supposed to be the defects for the code. The branch coverage can have more complexity if the code is changed to encounter defects. If branch coverage can find, then we say we have good test suits.

Between M1, M3 and M6: There is a co-relation as we can see that from the metric 6, as the size of the code base i.e. as the number of lines increases the test coverage will generally decrease as it becomes increasing daunting to have more coverage as the LOC increases by a large factor and hence the number of defects go up.

Between M1, M2 and M4:
Source code with high value of cyclomatic complexity contains a greater number of linearly independent path. Hence as the value of cyclomatic complexity increases we need a greater number of test cases for 100% statement and branch coverage. For both statement and branch coverage, we need to find paths that go through all statements and branch.

Between M5 and M6: There is less correlation between metric 5 and metric 6. The metric 5 is focused on the maintaining the source code after the deployment and metric 6 is about finding the defect density. The source code and predicates may lead to find some logical defects.

2. Related Work

The paper talks about various metrics for ascertaining code coverage. The paper expresses that there is a low to direct connection amongst coverage and effectiveness when the quantity of experiments in the suite is controlled for. It says that more grounded types of coverage don't give more noteworthy knowledge into the adequacy of the suite. Their outcomes propose that coverage, while helpful for distinguishing under-tested pieces of a program, should not be utilized as a quality objective since it's not a good indicator of test suite effectiveness.

This paper comprises the examination of a lot of huge open-source Java projects and measures the code coverage of the experiments that join these projects. The developers gather genuine bugs logged in the issue tracking system after the arrival of the product and investigate the connections between code inclusion and these bugs.

They additionally gather different metrics, for example, cyclomatic complexity and lines of code, which are utilized to standardize the number of bugs and coverage to correspond with different measurements just as utilize these metrics in regression analysis. Their outcomes show that coverage has an inconsequential relationship with the number of bugs that are found after the release of the software at the project level and no such connection at the document level. The motivation to select the metrics 5 is to get the perfect maintainability statistics. Often, we find that having variable names which has meaning has less likely to be defected. For metrics 6, if we have defect density of all the module we can judge and focus on highly likely defective module and give more attention to that module.

We will use various tools as we come to know the necessities in the long run, for example, EclEmma, Jacoco, etc. We are exploring more options for such tools.

3. Selected Open-Source System

| Open-source Project | Version | Line of Code | VCS | Issue Tracking |
|---|---|---|---|---|
| **Apache Common Collections** | 4.3 | 132K | ✓ | ✓ |
| **Apache Common Math** | 3.5 | 186K | ✓ | ✓ |
| **Apache Common Configuration** | 2.4 | 847K | ✓ | ✓ |
| **Apache Commons IO** | 2.6 | 35K | ✓ | ✓ |

These tasks have a high volume of clients and submit. A few engineers have taken a shot at these ventures which give a decent base to begin our examination on various measurements which we have contemplated. These undertakings have a few bugs and highlights revealed which give the knowledge to comprehend the measurements. They have numerous form discharges that assist with understanding the framework better with stable forms.

4. Resource Planning

We have divided the whole project in several tasks, which are as follows,

T1 - Literature review
T2 - Supporting Documentation
T3 - External tools
T4 - Metric Study
T5 - Install and Build
T6 - Compiling and Documentation

| Name | T1 | T2 | T3 | T4 | T5 | T6 |
|---|---|---|---|---|---|---|
| Hetvi | ✓ | | | ✓ | ✓ | ✓ |
| Khyatibahen | ✓ | ✓ | | ✓ | | ✓ |
| Sarvesh | ✓ | ✓ | | ✓ | | ✓ |
| Satish | ✓ | | ✓ | ✓ | ✓ | |
| Venkat Mani Deep | ✓ | | ✓ | ✓ | ✓ | |

| | |
|---|---|
| **Metric 1** | Hetvi and Khyatibahen |
| **Metric 2** | Hetvi and Khyatibahen |
| **Metric 3** | Manideep and Khyatibahen |
| **Metric 4** | Sarvesh and Satish |
| **Metric 5** | Sarvesh and Manideep |
| **Metric 6** | Satish and Manideep |

5. References

Code Coverage:
https://www.guru99.com/code-coverage.html
https://www.eclemma.org/
https://www.sealights.io/test-metrics/code-coverage-metrics/

Software Science Effort:
http://www.dmi.usherb.ca/~frappier/Papers/tm2.pdf

Cyclomatic Complexity:
https://en.wikipedia.org/wiki/Cyclomatic_complexity
https://www.eclemma.org/jacoco/trunk/doc/counters.html

Mutation Testing:
https://www.guru99.com/mutation-testing.html

Test suite effectiveness:
http://www.linozemtseva.com/research/2014/icse/coverage/coverage_paper.pdf

Michura, J., Capretz, M., & Wang, S. (2013). Extension of Object-Oriented Metrics Suite for Software Maintenance. ISRN Software Engineering, 2013, 1-14.

H. Zhang, "An Investigation of the Relationships between Lines of Code and Defects," IEEE.

J. Hayes, S. Patel and L. Zhao, "A metrics-based software maintenance effort model," IEEE, 2004.