+PROJECT PROPOSAL

1.Team Information

Team P

Name	Student ID	Email
Pritpal Kaur	40049323	94pritpalkaur@gmail.com
Sehajpreet Singh Gill	40088155	iamsherrysingh@gmail.com
Kartik Nagpal	40049792	kartiknagpal28@gmail.com
Karamveer Kaur Sangha	40047006	karamveerkaur28@gmail.com
Vrind Gupta	40092902	vrind.gupta1@gmail.com
Koteswara Rao Panchumarthy	40084998	kotichowdary18@gmail.com

2.Selected Metrics and correlation analysis

Metric	Detail	Metric Used
Metric 1	Test Coverage Metric	Statement Coverage
Metric 2	Test Coverage Metric	Branch Coverage
Metric 3	Test-suite effectiveness metric	Mutation Score
Metric 4	Complexity metric	McCabe complexity
Metric 5	Software maintenance metric	Code Churn
Metric 6	Software quality metric	Post-release defect density

A. **Metric 1 (Test Coverage Metric): Statement Coverage**

It is a white-box testing technique which pertains the number of statements covered in the code. This technique involves the execution of all statements of the source code at least once. It is used to calculate the total number of executed statements in the source code out of total statements present in the source code. It is a white-box testing technique which pertains the number of statements covered in the code. [1]

$$Statement\ Coverage\ = \frac{Number\ of\ executed\ statements}{Total\ number\ of\ statments} \times 100$$

B. **Metric 2 (Test coverage metric): Branch Coverage**

A white-box testing technique which ensures that every possible branch from a decision point is executed at least once. It covers all the possible outcomes (true and false) of each condition of decision point at least once. Branch coverage technique is a white box testing technique that ensures that every branch of each decision point must be executed.[2]

$$Branch\ Coverage\ = \frac{Number\ of\ executed\ branches}{Total\ number\ of\ branches} \times 100$$

C. **Metric 3 (Test-suite effectiveness metric): Mutation Score**

In this method of software testing, we test the effectiveness of test suites by mutating the code statements and verify that test cases are able to find errors. Test suits having high mutation score can detect more real fault as compared to test suit with low mutation score.

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

D. **Metric 4: Complexity Metric(McCabe complexity)**

Cyclomatic complexity is a source code complexity measurement that is being correlated to a number of coding errors. It is calculated by developing a Control Flow Graph of the code that measures the number of linearly-independent paths through a program module.[13]

Lower the Program's cyclomatic complexity, lower the risk to modify and easier to understand. It can be represented using the below formula:

$$Cyclomatic\ complexity = E - N + 2*P$$

E = number of edges in the flow graph.
N = number of nodes in the flow graph.
P = number of nodes that have exit points

E. **Metric 5: Code Churn**

Code churn is defined as lines added, modified or deleted to a file or a system from one version to another.

Code churn 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.[6].

Code churn can be a good metric in the phase of software maintenance as it measures the difference between two versions of a system.

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

The post-release Defect Density is the number of identified defects found during the operational phase per 1000 source lines of code. The post-release Defect Density is a quality indicator for Product Quality. Only the total number of added or modified source lines of code is used by the calculation of the post-release Defect Density[3].

The defect density zero indicates that the product contains no defects and also the best possible product quality. If the defect density increases the product quality decreases which means there are more defects per 1000 source lines of code.

The formula for calculating post-release defect density is

Let Defect Density $DD_{Post-release} = \frac{|D_{Post-release}|}{KSLOC}$ where

- $|D_{Post-release}|$: is the total number of post-release defects, where $D_{Post-release}$ is the total set of found post-release defects in the first month after delivery into the production environment.
- $KSLOC$: are the total number of added or modified $SLOC$, where $SLOC$ are the total number of 1000 source lines of code.

Figure 1: Formula for post-release defect density[3]

Correlation Analysis:

- **Relation between Test Coverage Metric (M-1 & M-2) and Test-Suite effectiveness (M-3)**:

Mutation Testing generates different versions (mutants) of a program under test by introducing small changes that are supposed to be defects in the code and we know that the statement and branch coverage criterion requires that all control transfers in the program under test are exercised during testing. More coverage tend to define the effectiveness.

Some papers [4] regarding this, evaluated the relationship between test suit size, test suit coverage, and its effectiveness i.e. researchers measured branch coverage, decision coverage and used mutation tests on modified condition coverage to analyze the test suit's effectiveness.

The conclusion states that though test coverage identify the under tested part of the system but it may not be used for the evaluation of the effective test suite.

- **Relation between Test Coverage Metric(M-1 & M-2) and Complexity Metric (M-4):**
  Higher cyclomatic value tends to suggest greater number of the independent paths and hence lower test coverage. Hence as the value of cyclomatic complexity increases we need a greater number of test cases for 100% statement and branch coverage [7].

- **Relation between Test Coverage Metric (M-1 & M-2) and Software Quality Metric (M-5):**

Classes with low test coverage (considering both statement coverage and branch coverage) contain more bugs is the rationale we are defining. 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 [7] [8].

- **Relation between Software Maintenance(M-5) and Software Quality Metric (M-6):**
  Software with a high code churn will contain more bugs, since a high code churn indicates software instability. Therefore, the software is likely to have a higher post-release defect density.

3. Related Work:

3.1 Metric 1 and 2 : Test Coverage Metrics - Branch coverage and statement coverage

According to the paper *Software reliability growth with test coverage*, test coverage metrics quantify the degree of the thoroughness of testing [14]. There is a correlation between Test coverage and software reliability - more test coverage indicates more reliable software. This paper presents the relationship between test timings, test coverage, and software reliability. The relationship between test coverage(branch and statement coverage) and defect coverage was found out using a LE model (Logarithmic exponential model) in the study mentioned in this paper. The LE model is a probabilistic model - it is based on the hypothesis that there will be a probability of exercising any enumerable element in code like statement or branch, just like there is a probability of defect being encountered. The results of the study conducted in this research concludes that relative defect density declines as the branch and statement coverage increases [14].

3.2 Metric Test Suit Effective - Mutation Score

Mutation Testing is a type of software testing where we mutate (change) certain statements in the source code and check if the test cases are able to find the errors. It is a type of white box testing which is mainly used for Unit Testing. The changes in the mutant program are kept extremely small, so it does not affect the overall objective of the program[5]. In the paper[9], authors investigated the underlying mutation operators of the mutants that are coupled to real faults when statement coverage did not increase. We found that real faults were more often coupled to mutants generated by the conditional operator replacement, relational operator replacement, and statement deletion mutation operators. A possible explanation is that some of these mutants cannot be detected by tests that only satisfy statement coverage. Conditional and relational operator replacement mutants are frequently generated within conditional statements, and numerous statement deletion mutants only omit side effects — detecting those mutants requires more thorough testing. Their study also suggests that this correlation is stronger than the correlation of the statement coverage and real fault detection ability of the test suit. Several researchers have studied the relationship between real faults and mutants. [9]

Methodology:

Input: 1. Code to be analyzed.

2. Mutant code with faults included.

3. Test cases for analyzing the faults.

Output: Mutation score of the mutant code when run along with the test cases [5].

Step 1: Enter the correct SLOC.

Step 2: Create Unit Tests for that SLOC.

Step 3: Create a mutation of the SLOC provided above with some faults introduced in it.

Step 4: Run the Unit Tests with the mutated code and check for the errors in the code.

Step 5: Calculate the Mutation Score from the formula:

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

Step 6: **If mutation score= 0%**, the test cases are not written correctly on the contrary if **mutation score=100%** means that all the faults are recognized completely [5].

3.3 Metric 4: Cyclomatic Complexity:

According to paper [11] & [12], Cyclomatic complexity give the measure of the complexity of program by measuring the linearly independent execution path through the piece of source code. It determines test cases which are necessary to achieve branch coverage.

Principles to consider the complexity metric (according to the paper citing):

- Complexity and program size relationship is non-linear.

- Complexity of program will be high, if more complicated data structures and/or control structures are used.
- Tight coupling between modules or procedures make program more complex.
- More global variables or non-local variables make program more complex.

The study hypothesized that, Complexity metrics can help to evaluate the cost of development. Complexity metrics can also be used to foresee the defects or errors.

They choose arbitrary CM1 dataset that contains 505 modules written in C Code, and study the Correlation Between Cyclomatic Complexity metric with number of lines of code and number of errors. They showed that the correlation of cyclomatic complexity with line of code is strong and the change in the number of line of code will impact on cyclomatic complexity metric.

3.4. Metric 5 & 6 : Code Churns and Defect Density

In the research paper **Use of relative code churn measures to predict system defect density** it was concluded that

1. Increase in relative code churn measures is accompanied by an increase in system defect density;
2. Using relative values of code churn predictors is better than using absolute values to explain the system defect density;
3. Relative code churn measures can be used as efficient predictors of system defect density

Analysed the code churn between the release of W2k3 and the release of the W2k3 Service Pack 1 (W2k3-SPl) to predict the defect density in W2k3-SPl.The relative code churn measures are statistically better predictors of defect density than the absolute measures.

<u>Points:</u>

1. Code churns is a better way to predict system's defect density rather than absolute values.
2. Code churns directly proportional to the measure of defect density.
3. Relative code churns is an efficient predictor of system defect density.
4. Relative code churns and is a better way to discriminate between faulty and non faulty binaries.

<u>Methodology used:</u>

They represented absolute code measures for data collection and changed them into their relative code churn measures.

Absolute metrics that were used are Total LOC, Churned LOC, Deleted LOC, File count, Week of churn, Churn count, file changed and their relative code churn metrics were

M1: Churned LOC / Total LOC

M2: Deleted LOC / Total LOC
M3: Files churned / File count
M4: Churn count / Files churned
M5: Week of churn / file count
M6: Lines worked on / Weeks of churn
M7: Churned LOC / Deleted LOC
M8: Lines worked on / Churn Count

Correlation between code churns and defect density Image 1 [11] shows the Spearman rank correlation among the defects/KLOC and the relative code churn measures. Spearman rank correlation is a commonly-used robust correlation technique[12].
"Thus, with an increase in the relative churn measures there is a corresponding positive increase in the defects/KLOC. This is indicated by the statistically significant positive Spearman rank correlation coefficient" [11]

4. Selected Open- Source Systems

Below is the list of projects among which 5 Projects will be considered for our Analysis purposes.

4.1 Apache Commons Collection

- Commons Collections package contains types that extend and augment the Java Collections Framework.
- **SLOC:**126.68k
- **Website:** http://commons.apache.org/collections/
- **GitHub:** https://github.com/apache/commons-collections.git
- **Version**: Apache Commons Logging 4.3
- **Issue Tracking System:**

https://issues.apache.org/jira/projects/COLLECTIONS/issues/COLLECTIONS-714?filter=allopenissues

4.2 Apache Commons Logging

- A thin adapter allowing configurable bridging to other, well-known logging systems
- **SLOC:** 9.63K
- **Website:** http://commons.apache.org/logging/

- **GitHub:** https://github.com/apache/commons-logging.git
- **Version**: Apache Commons Logging 1.2
- **BugTrackingSystem:** **https://issues.apache.org/jira/projects/LOGGING/issues/LOGGING-165?filter=allopenissues**

4.3 Apache Commons Configuration

- The Commons Configuration software library provides a generic configuration interface which enables a Java application to read configuration data from a variety of sources.
- **SLOC-** 105k
- **WEBSITE:** https://commons.apache.org/proper/commons-configuration/
- **GitHub:** https://github.com/apache/commons-configuration
- **Version:** Apache Commons Configuration 2.4 SnapShot

4.4 Apache Commons File Upload

- Provides a simple and flexible means of adding support for multipart file functionality servlets and applications.
- **SLOC:** 14.56k
- **Website:** http://commons.apache.org/fileupload/
- **GitHub:** https://github.com/apache/commons-fileupload.git
- **Version**: Apache Commons FileUpload 1.4
- **Issue Tracking System:** **https://issues.apache.org/jira/projects/FILEUPLOAD/issues/FILEUPLOAD-291?filter=allopenissues**

4.5 Apache Commons Math

- a library of lightweight, self-contained mathematics and statistics components addressing the most common practical problems not immediately available in the Java programming language or commons-lang.
- **SLOC:** 186K
- **Website:** http://commons.apache.org/math/
- **GitHub:** https://github.com/apache/commons-math.git

- **Version**: Apache Commons Math 4.0
- **Bug Tracking System:**
  https://issues.apache.org/jira/projects/MATH/issues/MATH-1462?filter=allopenissu

5. Tools and Techniques Used

A. **Metric 1:**

Jacoco Emma: a free code coverage library for Java

B. **Metric 2:**

Jacoco Emma: a free code coverage library for Java

C. **Metric 3:**

PiTest: It is a mutation testing system, providing coverage for Java and the jvm.

D. Metric 4:

Eclipse metrics plugin, SonarQube, Jacoco

E. **Metric 5:**

Eclipse metrics plugin, CLOC command line tool

There are many tools we can use for such analysis, however, a few tools that we found to be very useful are listed below [16]:

Git – code repository. We wrote a small tool that will connect to the git repository and pull the metrics for the given date period from the git repository range, like

a. List of files changed
b. Number of lines added / modified
c. Total code churn

N-Cover — This is a comprehensive tool to identify code coverage. We will be able to identify how much of the code is tested through tests executed and any gaps in the testing. It adds more value to figure out the gaps in testing and filling those gaps immediately.

F. **Metric 6:**

Discriminant Analysis: Discriminant analysis [15] is a statistical technique used to categorize elements into groups based on metric values which has been used to detect fault-prone programs.

PREfix: The PREfix tool[15] symbolically executes select paths through a C/C++ program. During this symbolic execution it looks for a multitude of common low-level programming errors, including NULL pointer dereferences, the use of uninitialized memory, double freeing of resources, etc.

PREfast: The PREfast tool[15] is a “fast” version of the PREfix tool. Its development was independent of the PREfix tool but aimed at desktop deployment. As a result, the PREfast analyses are inexpensive, accounting for negligible percentage of compile time.

6. Resource Planning

Team Member	**Responsibilities**
Vrind Gupta	Collection of Metric 1 results, documentation and system setup.
Sehajpreet Singh Gill	Collection of Metric 2 results, documentation and system setup.
Kartik Nagpal	Collection of Metric 3 results and documentation
Pritpal Kaur	Collection of Metric 4 results and documentation
Karamveer Kaur Sangha	Collection of Metric 5 results, Tool Installation and documentation.
Koteswara Rao Panchumarthy	Collection of Metric 6 results, Tool Installation and documentation.

REFERENCES

[1]https://www.javatpoint.com/statement-coverage-testing-in-white-box-testing
[2]https://www.javatpoint.com/branch-coverage-testing-in-white-box-testing
[3]https://research.infosupport.com/wp-content/uploads/2017/08/MasterThesis-LammertVinke-Final.pdf
[4] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In <em>Proceedings of the 36th International Conference on Software Engineering</em> (ICSE 2014). ACM, New York, NY, USA, 435-445. DOI: https://doi.org/10.1145/2568225.2568271
[5] Guru99, "Mutation Testing in Software Testing: Mutant Score & Analysis Example," *https://www.guru99.com/mutation-testing.html,* 2019.

[6]Use of relative code churn measures to predict system defect density. https://ieeexplore.ieee.org/document/1553571 .

[7] M.M.S. Sarwar, S. Shahzad and I. Ahmad, “Cyclomatic Complexity: The nesting problem,” IEEE,2014

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

[9] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?
https://doi.org/10.1145/2635868.2635929

[10] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are mutation scores correlated with real fault detection?: a large scale empirical study on the relationship between mutants and real faults.
https://doi.org/10.1145/3180155.3180183

[11] “The Correlation among Software Complexity Metrics with Case Study”, Tashtoush1, -Maolegi2, Arkok3 Yahya Mohammed Al-Bassam https://www.researchgate.net/publication/264936702_The_Correlation_among_Software_Complexity_Metrics_with_Case_Study

[12] “The Research on Software Metrics and Software Complexity Metrics”, Tu Honglei1, Sun Wei1, Zhang Yanan1 College of Information Technology, Beijing Normal University, Zhuhai 519086, 1 helentu_2006@163.com https://ieeexplore.ieee.org/document/5385114

[13] https://www.tutorialspoint.com/software_testing_dictionary/cyclomatic_complexity.htm

[14] Software reliability growth with test coverage.
https://ieeexplore.ieee.org/document/1044339

[15] “Static Analysis Tools as Early Indicators of Pre-Release Defect Density” , Nachiappan Nagappan Department of Computer Science North Carolina State University Raleigh, Thomas Ball Microsoft Research Redmond.p

[16] https://dzone.com/articles/code-churn-a-magical-metric-for-software-quality