Project Proposal

1. Team Information:

Team I

Team Member Names	Student ID	Contact Email
Chetan Paliwal	40083388	Chetanpaliwal22@gmail.com
Himen Hitesh Sidhpura	40091993	himens72@gmail.com
Sandeep Siddaramaiah	40087428	sandeepsiddaramaiah@gmail.com
Karthik B P	40094485	karthikbeepi@gmail.com
Rohan Deepak Paspallu	40093648	Paspallu.rohan@gmail.com

2. Selected Metrics and Correlation analysis

Metric 1: Statement Coverage

Metric 2: Branch Coverage

Metric 3: Mutation Score

Metric 4: Cyclomatic Complexity (McCabe)

Metric 5: Adaptive Maintenance Effort Model (AMEffMo)

Adaptive Maintenance Model help us to predict maintenance effort required in terms of person & hours.

Metric 6: Software Defect Density

Defect Density (DD) - defined as the number of defects divided by size - is often used as a related measure of quality.

2.1 Detailed Explanation of Metrics:

2.1.1 Statement Coverage and Branch Coverage

Objective:

Code coverage analysis is the process of finding portions of a program not used by a set of test cases, thereby resulting a quantitative measure of code coverage, which is an indirect measure of code quality [1]. It can also help in creating additional test cases to increase the coverage and may identify unreachable portions of the code. Additionally, it can identify redundant test cases that do not increase coverage, as well as help in testing changes made to the code during regression testing [2].

Hypothesis:

Coverage measures based on various code-elements such as methods, statements, blocks, branches, predicates are most widely employed for coverage-based testing. The coverage analysis tools are language

dependent. Coverage analyzers work by adding probe instructions in the program which increment counters [2].

Methodology:

Input: The java program to be analysed and their test-cases.

Output: The statements and branches covered in the test-cases.

Step 1: Construction of flow graph

Step 2: Initial path selection

Step 3: Derivation of linear constraints

Step 4: Detection of infeasible paths

Step 5: Consistent subset of linear constraints

Step 6: Path switching

Results:

The results produced by the branch and test coverage can be used to provide valuable input to the test-cases run for the program [1]. It can help the team responsible for testing provide more test cases so that the branch and statement coverage is 100% to be sure that everything in the given java program was properly tested and documented [2].

2.1.2 Mutation Score:

Objective:

Mutation testing is a type of testing in which we try to make changes to the source code so that we can check whether the test cases can find the errors in the code or the code runs without any errors [3]. This type of testing is used only for the Unit Testing methods to make sure that each part of a source code is properly tested [3].

Mutation Score is calculated as the percentage of the ratio of number of mutants killed by the total number of mutants in the SLOC [3].

- Mutation Score = (Killed Mutants / Total number of Mutants) * 100

Hypothesis:

Mutation Score is based on several code elements wherein the SLOC is taken and a test case is designed according to it so that the code runs without any anomaly [3]. Then the code is changing to introduce a fault in the system so that we can check whether the test case is able to site any faults in the source code [3]. 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%** [3]**.**

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 [3].

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 = (Killed Mutants / Total number of Mutants) * 100

Step 6: If mutation score= 0% the test cases are not written correctly on the contrary mutation score=100% means that all the faults are recognized completely [3].

2.1.3 Cyclomatic Complexity (McCabe):

Objective:

Cyclomatic complexity is a quantitative measure of the number of linearly independent paths through program's source code. Cyclomatic complexity is used as a benchmark to compare two different source code [4]. The program with high cyclomatic complexity is more error prone and require more understanding for testing. It also helps us in determining the number of test cases that will be required for complete branch coverage [4].

Cyclomatic complexity is calculate with the help of number of edges(E), number of nodes(N) and number of connected point(P) [4].

- Cyclomatic Complexity =E – N + 2P

Cyclomatic complexity can also be determined with the help of number of control predicate (D):

- Cyclomatic Complexity = D + 1

Hypothesis:

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 [4]. In general, a McCabe complexity of low is good to have, A high complexity (>10) makes the method more complex. A large switch statement can be clear to understand but in result it will give very high count of Cyclomatic complexity [4].

Methodology:

Input: The source Code to be analyzed.

Output: Cyclomatic complexity of the method/class.

Step 1: Start with a count of 1 for each method.

Step 2: Increment the count for each of the following element found in the source code:

- Selection: if, else, case, default.

- Loops: for, do-while, while, break and continue.
- Exception: catch, finally, throw.
- Operators: &&, ||.
- Returns: Each return statement which the last statement of the method is not.

Step 3: Determine the class complexity by adding complexity of each individual method.

Conclusion:

Complexity can have many meanings. It is used as a benchmark for predicating cost. It is also used to determine the number of test cases [4]. Cyclomatic complexity cannot be used efficiently in doing the code comparison to determine the code efficiency, two code with same number of control predicate can have different complexity, A nested loop with a billion iteration will have more computational complexity then a loop with hundred iteration [4].

2.1.4 Adaptive Maintenance Effort Model (AMEffMo)

Objective:

This model aims on calculating maintenance effort in terms of person-hours. There is various metric which are found to be strongly correlated to maintenance effort [5]. This metrics can be number of lines changed and number of operators changed.

Hypothesis:

This model hypothesis that maintenance effort for a software depends on measurable metrics derived from software development process [5]. In this model, metrics that affects estimation effort required for maintaining project is identified first. Then, correlation is established between identified metric and maintenance effort.

Metrics:

Table 1: Metrics and their description

	Metric	Description
1	%Operators Changed	Percent difference in total number of operators in the application after maintenance
2	LOC Difference (DLOC)	Lines of code edited, added or deleted during maintenance
3	% Mod change/add	% Code modules changed during Maintenance
4	Noprtr	Total number of operators
5	CF	Coupling factor
6	CR	Comment Ratio
7	Hdiff	Halstead's difficulty
8	LCOM	Lack of cohesion in methods
9	AC	Attribute Complexity
10	CC	Cyclomatic Complexity
11	TCR	True Comment Ratio
12	PM	Perceived maintainability
13	MP	Maintainability product
14	Classes Changed	Number of classes modified
15	MI	Welker's Maintainability Index
16	HPVol	Halstead program volume
17	Classes Added	Number of classes added
18	Heff	Halstead's effort
19	LOC	Total lines of code

Methodology:

Step 1: Identify metric which affect estimation effort

Step 2: Perform simple regression.

Step 3: In Regression, use datapoint collected from data source and use least square method to produce the following model: **E = -40 +6.56 DLOC**

Step 4: Use another variable number of operators changed (DNoprtr), to generate following model: **E = -124 +7.5 DNoprtr**

2.1.5 Software Defect Density

Objective:

It is always desirable to understand the quality of a software system based on static code metrics. In this paper, they analyze the relationships between Lines of Code (LOC) and defects (including both pre-release and post-release defects). They claim that they can use defect density values calculated from a small percentage of largest modules to predict the number of total defects accurately.

Methodology:

Step 1: Find the defect density using the following formula.
Defect Density = Number of defects/KLOC
KLOC = Number of lines of code in thousand.

Step 2: To improve LOC's ability in defect prediction using defect density following formula can be used. We compute the defect density for the top k% largest modules (dd_k%) as: (the number of defects the top k% largest modules contain) / (the total KLOC of the top k% largest modules) * 100%.

Step 3 : We also evaluate the prediction accuracy using the MRE (Magnitude of Relative Error) measure, which is computed as (100% * |Predict-Actual|/Actual).

2.2 Co-Relation Analysis:

- **Between Statement Coverage (M-1) and Mutation Score (M-3):**

The more statements are covered for Mutation score i.e. the more statements we change from the source code to do mutation testing, better is the test suite effectiveness.

- **Between Branch Coverage (M-2) and Mutation Score (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 branch coverage criterion requires that all control transfers in the program under test are exercised during testing. Hence if branch coverage is more i.e. if all control transfers are tested then the bugs introduced through mutation testing will be detected there by increasing the test suite effectiveness.

- **Between Statement Coverage (M-1), Branch Coverage (M-3) and Software Defect Density (M-6):**

Classes with low test coverage (considering both statement coverage and branch coverage) contain more bugs is the rationale we are defining. 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 [4] [6].

- **Between Statement Coverage (M-1), Branch Coverage (M-2) and Cyclomatic Complexity (M-4):**

Source code with high value of cyclomatic complexity contains a greater number of linearly independent path. For both statement and branch coverage, we need to find paths (from start to end of flowchart) that go through all statements and branch. Hence as the value of cyclomatic complexity increases we need a greater number of test cases for 100% statement and branch coverage [4].

- **Between Metric 5 and Metric 6**

Defect density helps us to compare small and large file by normalizing number of defects to the amount of code review [7]. Variation in defect density has lot to do with file's "risk" in the system. With the help of Metric 5, we can calculate person-time which help us to calculate inspection rate and defect rate [7].

Inspection rate= (Lines of code reviewed) / (Total person-time)

Manager might insist on a slower inspection rate especially on stable branch, close to product release or core module, when everyone wants to be more careful with code change [7]. Also, inspection rate helps us to predict amount of time required to change code [7].

Defect Rate= (Number of defects) / (Total Person-Time)

Defect rate helps us to identify at which rate, reviewer uncover defects in code [7].

3 Projects

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

Projects	**Version**	**Lines of Code**
Apache Ant	1.10.5	235K
Apache Commons Loggin	1.2	9.63K
JMeter	5.0	344K
Apache Commons Lang	3.8.1	79.8K
Apache Commons Math	3.5	186K

These projects have high volume of users and commits. Several developers have worked on these projects which provide a good base to start our analysis on different metrics which we have studied. These projects have several bugs and features reported which provide insight to understand the metrics. They have many version releases which helps to understand the system better with stable builds.

4 Resource Planning.

Metric 1	Karthik and Himen
Metric 2	Karthik and Rohan
Metric 3	Rohan
Metric 4	Chetan and Sandeep
Metric 5	Himen
Metric 6	Sandeep and Chetan

5 References

[1] Atlassian Support, "Atlassian Support," *What is Code Coverage Analysis?,* 5 Feb 2008.

[2] Wikipedia, "https://en.wikipedia.org/wiki/Code_coverage," *Code Coverage,* 2018 December 2018.

[3] Guru99, "Mutation Testing in Software Testing: Mutant Score & Analysis Example," *https://www.guru99.com/mutation-testing.html,* 2019.

[4] M. M. S. Sarwar, S. Shahzad and I. Ahmad, "Cyclomatic complexity: The nesting problem," *IEEE,* 2014.

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

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

[7] SmartBear Software, "https://support.smartbear.com/collaborator/docs/reference/metrics.html".