**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](mailto:himens72@gmail.com) |
| Sandeep Siddaramaiah | 40087428 | sandeepsiddaramaiah@gmail.com |
| Karthik BP | 40094485 | karthikbeepi@gmail.com |
| Rohan 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.

**Co-Relation**

* **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 QMOOD (M-4):**

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.

* **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.

* **Between Metric 5 and Metric 6**

**3. Related Work**

* 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. 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.

**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.

**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. 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.

* 1. **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. 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.

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

* 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. Then the code is change 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. 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%.**

**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. (According to the formula of Mutation Score)

Step1: Enter the correct SLOC.

Step2: Create Unit Tests for that SLOC.

Step3: Create a mutation of the SLOC provided above with some faults introduced in it.

Step4: Run the Unit Tests with the mutated code and check for the errors in the code.

Step5: Calculate the Mutation Score from the formula:

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

Step6: 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.

**Results:**

The results produced by the Mutation score can be used to determine the amount of cases in the test cases which are not yet been recognized by the unit tests and are not included. This can be useful for creating better unit tests so that there is complete coverage of all the possible faults that can exist in a code.

The result produced by the mutation score can be used as an incremental method for the better development of the unit test cases for covering most of the unseen faults.

* 1. **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. 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.

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

* 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. 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.

**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. 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.

* 1. **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. 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. 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 operatorschanged (DNoprtr), to generate following model: E = -124 +7.5 DNoprtr

* 1. **Software Defect Density**

**Objective:**

In this paper, they collect and aggregate Defect Density (DD) figures published in literature, in addition we characterize DD as a function of different project factors in terms of central tendency and dispersion. The factors considered include development mode –open vs. closed source–, programming language, size, and age**.**

**Hypothesis:**

Defects are an ineludible component of software, Defect Density (DD) - defined as the number of defects divided by size - is often used as a related measure of quality. Project managers and researchers alike would benefit a lot from overview DD figures from software projects, the former for decision making the latter for state-of-the-practice assessment.

**Results:**

They were able to identify 19 papers reporting defect density figures concerning 109 software projects. The mean DD for the studied sample of projects is 7.47 post release defects per thousand lines of code (KLoC), the median is 4.3 with a standard deviation of 7.99. Development mode, is characterized by statistically meaningful different DD, the same for Java vs. C. Besides, in the studied sample large projects exhibited lower DD than medium and small projects.

**4. Projects**

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

|  |  |
| --- | --- |
| **Projects** | **Version** |
| Apache Ant | 1.10.5 |
| JRuby | 9.2.5.0 |
| JMeter | 5.0 |
| JFreeChart | 1.0.19 |
| BlueJ | 4.1.4 |
| Mockito | 2.23.20 |

These are open source software systems which are built using Java programming language. These projects fall under different ecosystems as Apache Ant is a software tool for automating software build processes, JRuby is a software used for development of Ruby programming language, JMeter is software used as a load testing tool for analyzing and measuring performance of a variety of services, Jfreechart is used for creation of wide variety of both interactive and non interactive charts, BlueJ is a IDE(integrated development environment) for the Java programming language, developed mainly for educational purposes, but also suitable for small-scale software development and Mockito is a testing framework used for the purpose of test-driven development or behavior-driven development.

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.

**6. 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 |

**7. References.**

[1] N. Gupta , A.P Mathur, M.L Souffa. 2000. Generating Test Data for Branch Coverage. In Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering. DOI: <https://doi.org/10.1109/ASE.2000.873666>.

[2]R. Lingampally, A. Gupta, P. Jalote. 2007. A Multipurpose Code Coverage Tool for Java. In proceedings 2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07). DOI: <https://doi.org/10.1109/HICSS.2007.24>

|  |  |  |
| --- | --- | --- |
| [3] |  | J. Hayes, S. Patel and L. Zhao, "A metrics-based software maintenance effort model," *IEEE,* 2004. |
| [4] |  | Mir Muhammd Suleman Sarwar, Sara Shahzad, Ibrar Ahmad, “Cyclomatic Complexity: The Nesting Problem” IEEE, 2013. |