# Dear Professor,

Our team would like to submit the report based on the following content selected project links and descriptions, metric descriptions, steps for collecting the data, steps for analyzing the data, describing the results (e.g., descriptive summaries for collected metrics, results of correlation analysis etc.), related work.

In this paper, we did correlation analysis among followed six metrics: branch coverage, statement coverage, mutation score, McCabe complexity, code churns and Code Smells.

The followed table is the information of our team members:

|  |  |  |
| --- | --- | --- |
| **Name** | **Student Number** | **E-mail** |
| Karan Sharma | 4080005 | 95sharma.karan@gmail.com |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Following is a link to the replication package in GitHub:

https://github.com/Shivamnautiyal20/SOEN6611\_Project\_Group\_C.git

Thank you,

Team C

**Project 1**

**Jfree Chart :**

JFreeChart is a comprehensive free chart library for the Java(tm) platform that can be used on the client-side (JavaFX and Swing) or the server side (with export to multiple formats including SVG, PNG and PDF).

JFree Chart is a maven project with the size of 167K LOC along with it contains 120k LOC of java code with continuous issue tracking on GitHub and sourceforge.net as well. We have selected mainly five version of code i.e. 1.0.14, 1.0.15, 1.0.16, 1.0.17, 1.0.19.

**Metrics Description :**

We have selected different metrics to measure the software and to analyse these software based on the values generated by each metric. These metrics provide different:-

**Metric 1 : Statement Coverage**

**Statement coverage** is a white box testing technique, which involves the execution of all the statements at least once in the source code. It is a metric, which is used to calculate and measure the number of statements in the source code which have been executed. It can also be used to check the quality of the code and the flow of different paths in the program.

Generally, based on the input given, some of the statements can be covered and some can not be executed. The goal of the statement coverage is that at least once all lines is covered at least once.

**Statement coverage = No of statements Executed/Total no of statements in the source code \* 100**

The Results we got are here-

|  |  |  |  |
| --- | --- | --- | --- |
| **System Name** | **Covered Statements** | **Missed Statements** | **Total Statements** |
| **Jfree Chart** | 526 | 113,481 | 365,983 |

|  |  |  |  |
| --- | --- | --- | --- |
| **System Name** | **CoveredBranches** | **Missed**  **Branches** | **Total Branches** |
| **Jfree Chart** | 124 | 13177 | 24040 |

**Metric 2: Branch Coverage**

**Branch coverage** is a testing method, which aims to ensure that each one of the possible branches from each decision point is executed at least once and thereby ensuring that all reachable code is executed. Statement coverage is essential, it also has some defects. For example, statement coverage only considers the executed statements and ignore the combinations of branches.

By using the Branch coverage method, you can also measure the fraction of independent code segments. It also helps you to find out which sections of code don't have any branches. That is, every branch taken each way, true and false. It helps in validating all the branches in the code making sure that no branch leads to abnormal behaviour of the application.

Branch Testing = (Number of decisions outcomes tested / Total Number of decision Outcomes) x 100%

**Metric 3:**

**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.Each mutated version is called a mutant and tests detect and reject mutants by causing the behaviour of the original version to differ from the mutant.

The number of mutants depends on the definition of mutation operators and the

syntax/structure of the software 100% mutation score means killing all mutants (or a random sample).

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

**Metric 4 Cyclomatic complexity (McCabe complexity)**

**McCabe complexity** is a software metric used to measure the complexity of a program. It is a quantitative measure of independent paths in the source code of the program.

An independent path is defined as a path that has at least one edge which has not been traversed before in any other path. Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program.

1. **Cyclomatic Complexity = E – N + 2P,** where

E = the number of edges in CFG

N = the number of nodes in CFG

P = the number of connected components in CFG

D = is the number of control predicate (or decision) statements

**b)   D + 1,** where

D = is the number of control predicate (or decision) statements

**Metric 5- Software Maintenance Metric :Code Churn**

Code churn is a means to figure out the changes in the code from one release to next. It monitors changes in code like refactoring , adding new features etc. to ensure that the code remains stable after these changes. In this metric, we mainly focus on how many number of lines have been changed to achieve the functionality[2]. It provides the information like :

* If it's a completely new code (class, method).
* Whether it requires unit-level tests to ensure it is tested for various inputs, output, and associated business rules.
* What existing code was modified for specific (new or modified) business rules, data changes, etc.

**Metric 6 - Code Smells**

Code smells are a set of common signs which indicate that code is not good enough and it needs refactoring to finally have a clean code [3].There are various code smells like duplicated code, Long Methods, Feature Envy, God Class etc. Smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality".[[5]](https://en.wikipedia.org/wiki/Code_smell#cite_note-5) Code smells are usually not [bugs](https://en.wikipedia.org/wiki/Software_bug); they are not technically incorrect and do not prevent the program from functioning. Instead, they indicate weaknesses in design that may slow down development or increase the risk of bugs or failures in the future. Bad code smells can be an indicator of factors that contribute to [technical debt](https://en.wikipedia.org/wiki/Technical_debt) [3].

Types :

* **God class**: a class that has grown too large.
* **Feature envy** : a class that uses methods of another class excessively and it resulted in high coupling which is against the design quality.
* **Long method** : A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.

1. **STEPS TO COLLECTING THE DATA**

We divide the data collecting process in to following steps:

* Step 1: Choosing the open source projects (three open-source systems having at least 100K SLOC)
* Step 2: Building the project and running the test cases.
* Step 3: Configuring JaCoCo - free code coverage library for Java.
* Step 4: Configuring PIT for Mutation Testing.
* Step 5: Configuring Metrics Reloaded plugin in IntelliJ Idea IDE for static code analysis.
* Step 6: Manual work for data cleaning and correlation calculation for metrics using MS Excel.

*Step1: Choosing the open source projects*

We have spent lot of time on finding and choosing the open source projects so that these projects meets the basics requirement of this research. We made sure to meet the following criteria:

* An open source project must have majority of Java codes and should have enough line of codes(SLOCs).
* It should have at-least 5 released versions with tracking system.
* It should have active and well managed issue tracking system e.g. on GitHub.
* It must have Junit test suits in order to collect data for mutation testing.
* It should have at-least some documentations that can guide us to build the project.

*Step 2: Building the projects and running the test cases.*

After choosing the projects, our first and main step was building them. We chose to use Intelli J and Eclipse as an IDE for building projects. There was a lot of difficulty building and running unit tests of these projects. It had some build errors due to various problems in each of the projects. Major problems include:

* Missing maven dependencies.
* Incompatible JDK version with that of installed in the system.
* Incompatible Junit version resulting in compilation errors.

After running through and solving these problems we moved to running unit tests and made sure that most of the unit tests are running correctly so that to make sure that running tests are available for collection of data related to all metrics selected.

*Step 3: Configuring JaCoCo -free code coverage library for Java.*

For statement coverage and branch coverage that are our required Metric 1 and 2. we selected JaCoCo library since it is a free and popular code coverage library for Java. Also, its plugin is also available in IntelliJ which is very easy to configure. IntelliJ Idea has built in JaCoCo configured. There are no additional dependencies need to be added. Following steps are required to run JaCoCo.

*Step 1:* Make sure the project is building and test cases are running successfully as in Figure 1.0.

*Step 2:* Create new Run/Debug configuration. Go to Run > Create/Edit configuration. As shown in Figure 1-1, choose “All in package” as test kind. Also make sure to choose the configuration for JaCoCo run as shown in Figure 1-2.

*Step 3:* Apply the newly created run profile and run JaCoCo with test coverage. Go to Run > “Run JaCoCo with Coverage”.

*Step 4:* Export the generated coverage results as shown in Figure 1-3.

A screenshot of a cell phone

Description automatically generated

Fig 1.0

A screenshot of a computer screen

Description automatically generated

Fig. 1.1

A screenshot of a cell phone

Description automatically generated

Fig 1.2

A screenshot of a cell phone

Description automatically generated

Fig 1.3

A screenshot of a cell phone

Description automatically generated

Results

*Step 4: Configuring PIT for Mutation Testing.*

Metric 3is about test suit effectiveness, and we chose PIT tool (<http://pitest.org>) to calculate mutation score. IntelliJ Idea has a plugin that simply provides results when we run mutation test on the selected test suits. The tool adds a 'Run configuration' that allows to execute PIT within IDE.

Usage: Run->Edit Configurations->Defaults->Pit Runner or just simply right click on project’s parent directory and choose to run PITest from the context menu as shown in Figure 2-1. Note here, it will apply the mutation on all the classes in the project since we are choosing the same option.

A screenshot of a computer

Description automatically generated

*Figure 2-1 PIT Configuration and running*

After completion of running the mutation testing, it will generate the test reports like shown in Figure 2-2.

A screenshot of a cell phone

Description automatically generated

*Step 5: Code Churn using CLOC library.*

For analysing the code churn we used CLOC library which provides the functionality for the comparison between five versions of the project.

The tool itself is very easy to use, we just need to install CLOC, specify the older and newer versions of the project along with the output file to print out the results as follows.

A screenshot of a cell phone

Description automatically generated

A screenshot of a computer

Description automatically generated

*Figure 3 CLOC configuration and usage*

*Step 6:* *Analyse code smells and collect data for different versions of the software.*

For the code smell, we basically used Jedeodorant plugin available in Eclipse. We decided to select latest five versions of the project. There are various code smells that are available in the code of these five versions as shown below. There are following steps that needs to be followed to use this plugin:

* 1. Install JDeodorant from Eclipse and restart Eclipse.

A screenshot of a social media post

Description automatically generated

* 1. In toolbar menu, select Bad Smell menu as shown follow and can use any kind of code smell we want to select.

A screenshot of a computer screen

Description automatically generated

* 1. Then, select the package we want to find code smells in and click on the “i” button and it will show results and we can export results to file as shown follows :

A screenshot of a social media post

Description automatically generated

* 1. We repeated same steps for all of the other code smells as well.
  2. The exported files are in .text format so we converted this data in excel format.

<http://www.jfree.org/jfreechart/>

<https://dzone.com/articles/code-churn-a-magical-metric-for-software-quality>

<https://en.wikipedia.org/wiki/Code_smell>