

**SOEN 6611**

**Software Measurement**

**Project Report**

ZUBIN MEHTA: 40049086

zubin.mehta07@gmail.com

GURSIMRAT KAUR: 40070868

gursimrat2359@gmail.com

SANDEEP SINGH: 40043110

sandeep.19920622@gmail.com

https://github.com/z-mehta/SOEN6611---Project/tree/master/SOEN6611%20-%20Project

***Abstract*—** Software testing is the crucial part of software development. Software testing activity ensures the quality of the program code as it evolves over time. It becomes important to perform adequate testing on the system with increasing size and complexity. Code coverage helps in evaluating the effectiveness of test suite which is one metric and other metrics like mutation score, Cyclomatic Complexity, Defect removal density and Post-release defect density which can also be considered to gauge the comprehensiveness of the test cases and adequacy of testing.

In this study we have analyzed three software systems with LOC ranging from 10 to 100K to find the correlation between the different metrics applied on these systems. We use eclipse plugin EclEmma to generate the code coverage to analyze if test suites can detect the real bugs or not. Mutation score is calculated by performing PI testing on each of the software system. We have used issue tracking system Github to calculate defect removal efficiency and post-release defect density. The results shows that there is a positive correlation between mutation score and code coverage but defect removal efficiency and post release defect density has a negative correlation between them.

***Keywords—***Statement and Branch coverage, Cyclomatic complexity, Mutation Score, Defect removal efficiency, Post release defect density.

# **Introduction**

Software products are hard to maintain. Software systems become large and complex hence their cost of maintenance is trivial. Testing the software to ensure the correctness of the behavior of the software becomes a vital activity. However, complete testing is infeasible. Effectiveness of testing depends upon the quality of test suites i.e. test suites must find all the faults in the system.

One of the metrics to calculate the effectiveness of test suites is coverage. Statement and branch coverage are the number of lines of code and branches executed by the test suite. Another approach to access the effectiveness of the test cases is mutation testing. In mutation testing the faults are seeded (mutants) in the source code and check whether the test suites are able to find the mutants or not. The correlation between the coverage and the mutation score is investigated. The experiment is performed on three open source system two of them with source code lines (LOC) greater than 100K. We have selected two large projects ImageJ and Checkstyle and one small open source system Checksum. The experimentation is performed on these systems and result is analyzed.

We used java code coverage tool EclEmma to measure the code coverage. Mutation score is calculated using eclipse plugin pitclipse and results shows strong correlation between coverage and mutation score. On the other hand, design complexity has also played an important role in the quality of the software system in development environment. Complexity metric have been used in attempt to find the defects in the system. McCabe IQ tool is used to evaluate the complexity of the chosen three systems. To determine the correctness of the software we have chosen the Defect removal efficiency (DRE) and Post-release defect density metric. Defect removal efficiency provides measure of the ability of the software development team to found and remove defects prior to the release whereas post-release defect density is the total number of defects divided by the size of the software. We have analyzed the result of DRE and DD measured on these three systems (ImageJ, Checksum, Checkstyle) and predicted that they have negative correlation between them.

1. **System Overview**

In our study, we have selected three open source systems that are ImageJ, Checkstyle and Checksum with their 8 versions. The system we have select is from the website OpenHub[2][3][4]. The metrics 1,2 and 3 are branch coverage, statement coverage and cyclomatic complexity. The 4th metric is to calculate the mutation score through mutation testing. For our study we have selected defect removal efficiency as 5th metric and for 6th metric it is post-release defect density. These metrics are applied on the systems and results and correlation between metrics is analyzed. However, each system is briefly described below:

In a Nutshell, ImageJ has had 2,068 commits made by 7 contributors representing 149,039 lines of code and is mostly written in Java with a low number of source code comments. It has a well-established, mature codebase maintained by a small development team with stable Y-O-Y commits. It took an estimated 39 years of effort (COCOMO model) starting with its first commit in July, 2008 ending with its most recent commit 6 months ago.

Whereas, Checkstyle has had 8,335 commits made by 235 contributors representing 264,005 lines of code is mostly written in Java with an average number of source code comments. It has a well-established, mature codebase maintained by a very large development team with decreasing Y-O-Y commits. It took an estimated 69 years of effort (COCOMO model) starting with its first commit in June, 2001 ending with its most recent commit 17 days ago.

Also, CheckSum has had 266 commits made by 14 contributors representing 4,929 lines of code and is mostly written in Java with a very well-commented source code has a well-established, mature codebase maintained by an average size development team with stable Y-O-Y commits. It took an estimated 2 years of effort (COCOCO model) starting with its first commit in July, 2010 ending with its most recent commit about 1 month ago.

# **Related Work**

We have studied many research papers and lots of literature available for metric 1,2 and 3. The one we find the best is **“***Teaching Software Testing Concepts Using a Mutation Testing Game”.* In this research paper they introduce CODE DEFENDERS game in which the players indulge in software testing techniques in a fun way. The concept of the game is based on the mutation testing. The players can act like attacker who’s aim is to break the code in a way that is not detected by the test cases. Mutation testing: modify (mutate) some statements in the code and check if test cases can find such defects (i.e., some test cases fail!). Addition to this statement and branch testing is also integrates in the Code Defender.

For Metric 5 we were able to find one literature related to DRE (Defect removal efficiency) which is the one of the most important and effective measure of quality metrics. To calculate DRE for a software/component we are using the formula provided in the literature. This paper shows how quality of software can be improved by rising the DRE to 99%. Hence Software defect potential is calculated for every phase of the software. For predicting the Defect density, we searched for the formula posted under metric 6. We are further searching for research papers to get good grasp of metrics and how to apply these metrics successfully to get the results.

# **Metric 1 & 2: Statement And Branch Coverage**

The code coverage metric is obtained using eclipse plugin EclEmma Java Code coverage with version 3.1.1.201809121651 by following the below mentioned steps:

**Step 1: Installing the plugin:**

The plugin is installed using eclipse marketplace by following the on-screen instructions

**Step 2: Executing the plugin:**

For each and every open system source code the tests were ran using eclipse as coverage.

**Minimum Requirement:**

* Java 1.7
* Junit 4.0
* EclEmma 3.1.1.201809121651

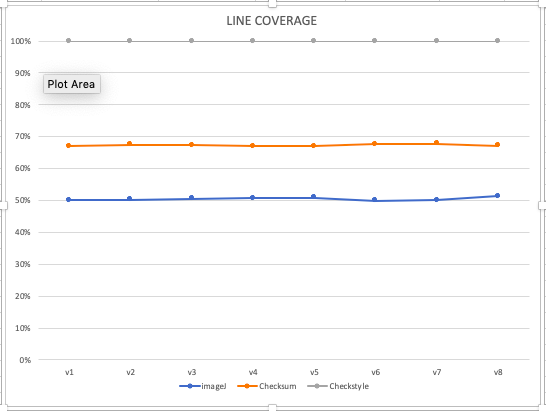
The following summary is obtained after running the test with line coverage shown in table 1.1, 1.2 and branch coverage table 2.1, 2.2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | V1 | V2 | V3 | V4 |
| ImageJ | 95.3% | 95.3% | 95.3% | 95.3% |
| Checksum | 32.2% | 32.8% | 31.7% | 30.6% |
| Checkstyle | 62.8% | 61.6% | 61.6% | 61.8% |

Table 1.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | V5 | V6 | V7 | V8 |
| ImageJ | 95.3% | 95.3% | 95.3% | 95.3% |
| Checksum | 30.2% | 33.9% | 33.8% | 29.6% |
| Checkstyle | 61.8% | 61.8% | 61.3% | 61% |

Table 1.2

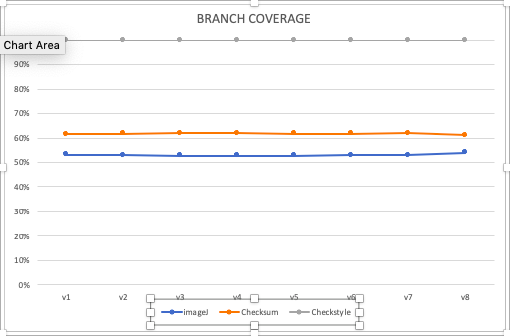


Line coverage Fig 1.a

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | V1 | V2 | V3 | V4 |
| ImageJ | 85.7% | 85.7% | 85.7% | 85.7% |
| Checksum | 13.6% | 14.2% | 14.9% | 15.1% |
| Checkstyle | 61.9% | 61.7% | 61.8% | 61.9% |

Table 2.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | V5 | V6 | V7 | V8 |
| ImageJ | 85.7% | 85.7% | 85.7% | 85.7% |
| Checksum | 14.6% | 14.4% | 14.4% | 11.7% |
| Checkstyle | 62.0% | 62.0% | 61.7% | 61.6% |

Table 2.2

Branch coverage Fig 1.b

# **Metric 3: Test Suite Effectiveness – Mutation Score**

The Mutation Score metric is used to determine the test suite effectiveness and it is calculated using eclipse plugin pitclipse with version 1.1.6.201607050705 by following the below mentioned steps:

**Step 1: Installing the plugin:**

The plugin is installed using eclipse marketplace by following the on-screen instructions

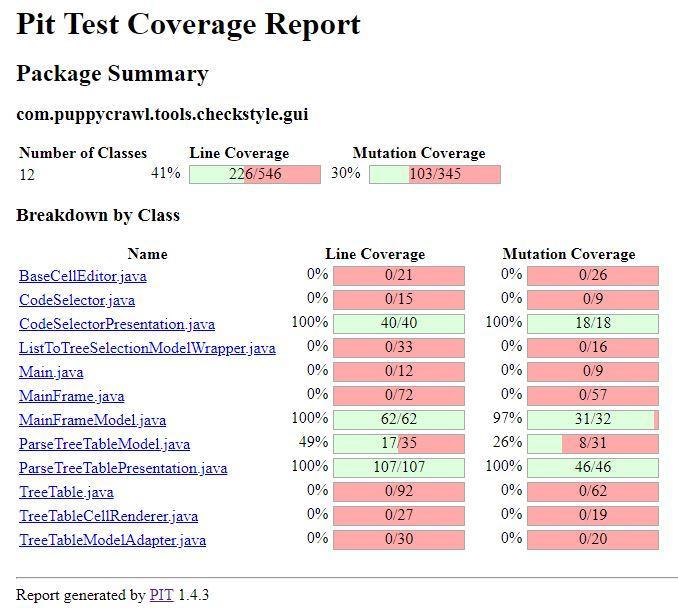
**Step 2: Executing the plugin:**

For each and every open system source code the tests were ran using eclipse as PIT Mutation.

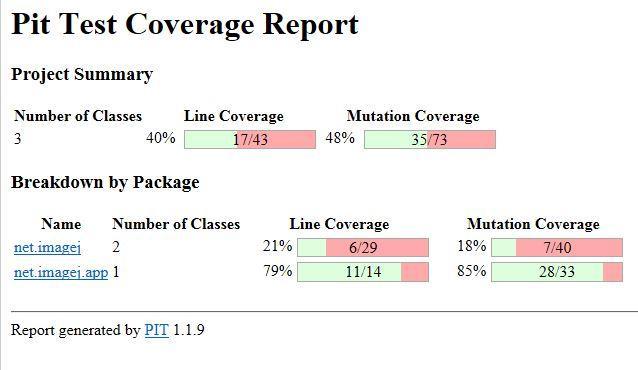
**Minimum Requirement:**

* Java 1.7
* Junit 4.0
* Pitclipse 1.1.6.201607050705

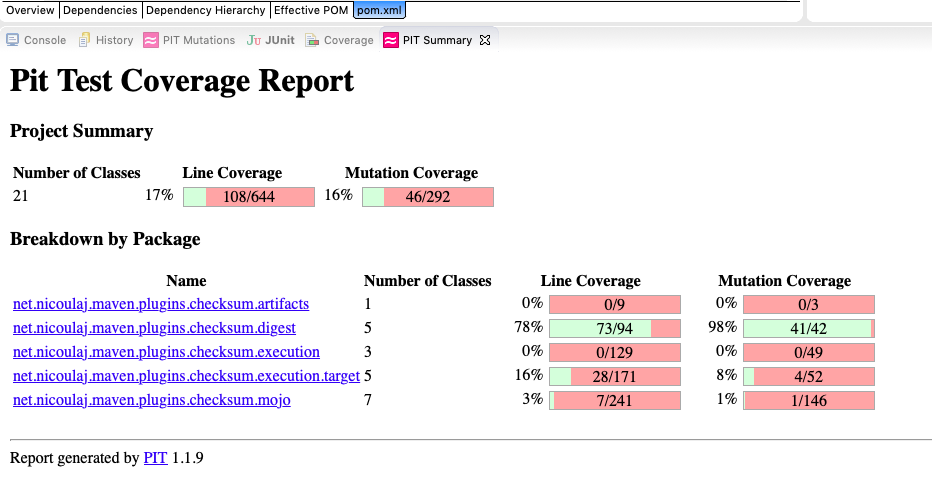
The following summary is obtained after running the test with coverage shown in Fig 2.1, Fig 2.2, Fig 2.3



Checkstyle Mutation Coverage Fig 2.1



Checksum Mutation coverage Fig 2.2



Imagej Mutation Coverage Fig 2.3

# **Correlation Between Metric 1,2 and 3**

The following items are some of the facts that can be inferred from these summaries shown in figures above:

* Strong test suites ensure the correctness and quality of the software.
* Coverage and mutation score has been greatly used to evaluate test suite effectiveness.
* The result shows that, the metric 1,2 and 3 has a positive correlation between them which implies that test suite with high coverage has high mutation score.
* The ImageJ has the highest coverage and mutation score than the checksum and checkstyle.
* It can be depicted from the results that, the ImageJ has the greater effectiveness of their test suites in comparison with other two systems.
* Checksum has the least branch coverage and therefore its mutation score is lowest.

# **Metric 4: McCabe Cyclomatic Complexity**

The McCabe cyclomatic complexity metric is used to deduce the code complexity and it is calculated using McCabe IQ tool with version 8.5 by following the below mentioned steps:

**Step 1: Import the source code:**

1. Start McCabe IQ Battlemap application and create a new project.
2. Select options as java programming language with respective version.
3. Select/Add hierarchies of your source code
4. Click ok

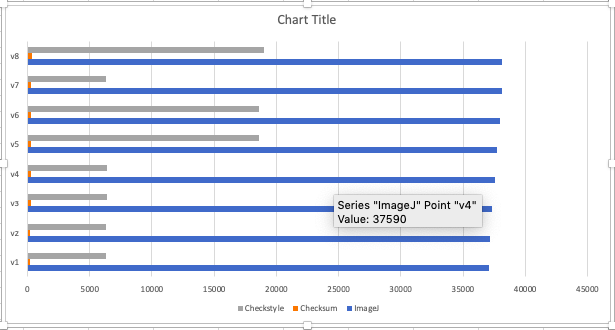
**Step 2: Execute the McCabe Project:**

After importing the source code, just run the application to obtain the complexity and publish the report of the system.

**Minimum Requirement:**

* Java 1.5
* McCabe v8.5

The following summary is obtained after running the McCabe IQ Battlemap shown in Table 3.1, Table 3.2



McCabe complexity graph with versions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | V1 | V2 | V3 | V4 |
| ImageJ | 37066 | 37165 | 37352 | 37590 |
| Checksum | 222 | 222 | 276 | 285 |
| Checkstyle | 6281 | 6313 | 6357 | 6357 |

Table 3.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | V5 | V6 | V7 | V8 |
| ImageJ | 37724 | 37932 | 38131 | 38111 |
| Checksum | 285 | 302 | 304 | 345 |
| Checkstyle | 18634 | 18591 | 6291 | 18988 |

Table 3.2

# **Correlation Between Metric 1, 2 and 4**

The following correlation can be derived from the results shown in Fig 3.1, 3.2, 1.1, 1.2, 2.1, 2.2.

* + Cyclomatic complexity is the measure of a number of linearly independent paths executed in the source code.
  + The number of test cases is equal to the number of independent paths.
  + Higher the number of branches in the source code, the larger is the complexity of the code.
  + Checkstyle has the 73279 lines executed with the cyclomatic complexity of 18988 which is higher than the checksum with 1224 lines executed and having a cyclomatic complexity of 345.

From the results, it can be deduced that the system with large coverage has a more cyclomatic complexity.

# **Metric 5: Defect Removal Efficiency (DRE)**

The defect removal efficiency metric is used to measured developer’s quality assurance efforts before release and it is the rate at which team members have been able to address and fix identified flaws in the program.

# **Defect Removal Efficiency (DRE) Algorithm**

By comparing the raw number of defects identified during the course of a development cycle with the number of flaws

actually repaired or eliminated, QA management can discern a team’s overall defect removal efficiency.

The defect removal efficiency metric consists of three steps as follows:

**Step #1**: Collect the raw material: You are going to need the number of defects found by the developers in development phase and the number of defects found by the user in the testing phase.

In this step, the number of defects in the development phase is calculated using an eclipse plugin known as Spot-bugs with version 3.1.12.r201903011242-190e1e at maximum analysis level for each open source system.

And the number of defects in the testing phase is calculated by counting issues raised by the users at the GitHub repository for respective version of each open source system prior to the release.

**Step #2**: Calculate the total defects, which is, sum of number of defects found in development phase by the developers and the number of defects found in testing phase by the user.

**Step #3**: Calculate defect removal efficiency using the formula mentioned below:

**Formula:**

1. **Defect removal efficiency** =

((#𝑫𝒆𝒗𝒆𝒍𝒐𝒑𝒎𝒆𝒏𝒕 𝑫𝒆𝒇𝒆𝒄𝒕𝒔 )/(𝒕𝒐𝒕𝒂𝒍 𝒅𝒆𝒇𝒆𝒄𝒕𝒔))**\*100**

**2**. 𝒕𝒐𝒕𝒂𝒍 𝒅𝒆𝒇𝒆𝒄𝒕𝒔 = #𝑫𝒆𝒇𝒆𝒄𝒕𝒔 𝒇𝒊𝒏𝒅 𝒅𝒖𝒓𝒊𝒏𝒈 𝒅𝒆𝒗𝒆𝒍𝒐𝒑𝒎𝒆𝒏𝒕 + #𝒅𝒆𝒇𝒆𝒄𝒕𝒔 𝒇𝒐𝒖𝒏𝒅 𝒃𝒚 𝒖𝒔𝒆𝒓𝒔

For this metric, we have considered eight versions of each system and maximum analysis level with high confidence level to find bugs using spotbugs plugin. This configuration will lead to obtain high accuracy, minimal change detection and effective results for defect removal efficiency.

The defect removal efficiency is obtained for each and every version of the respective system and then the mean efficiency is considered as overall defect removal efficiency for that particular system.

The results are shown below in table 4.1, 4.2, 4.3, 4.4 for each open source system with different versions:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | **Version 1** | | **Version 2** | |
| **No. of Defects**  **Found** | | **No. of Defects**  **Found** | |
| **During Development phase** | **By user** | **During Development phase** | **By user** |
| ImageJ | 3596 | 10 | 3596 | 10 |
| Checksum | 16 | 2 | 16 | 2 |
| Checkstyle | 8437 | 407 | 8746 | 414 |

Table 4.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | **Version 3** | | **Version 4** | |
| **No. of Defects**  **Found** | | **No. of Defects**  **Found** | |
| **During Development phase** | **By user** | **During Development phase** | **By user** |
| ImageJ | 2986 | 11 | 3605 | 12 |
| Checksum | 17 | 3 | 10 | 3 |
| Checkstyle | 8789 | 424 | 8797 | 426 |

Table 4.2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | **Version 5** | | **Version 6** | |
| **No. of Defects**  **Found** | | **No. of Defects**  **Found** | |
| **During Development phase** | **By user** | **During Development phase** | **By user** |
| ImageJ | 3632 | 12 | 3635 | 12 |
| Checksum | 10 | 4 | 10 | 5 |
| Checkstyle | 8805 | 445 | 8813 | 452 |

Table 4.3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **System Name** | **Version 7** | | **Version 8(Latest)** | | **Mean DRE**  **(Defect Removal Efficiency)** |
| **No. of Defects**  **Found** | | **No. of Defects**  **Found** | |
| **During Development phase** | **By user** | **During Development phase** | **By user** |
| ImageJ | 3648 | 13 | 3646 | 13 | 99% |
| Checksum | 15 | 5 | 16 | 7 | 77.2% |
| Checkstyle | 8903 | 464 | 9009 | 482 | 94.8% |

1. **Metric 6: Post- Release Defect Density (DD)**

The Post-release defect density metric is about software quality attribute and it is the number of confirmed defects detected in software/component during a defined period of development/operation divided by the size of the software/component.

# **Post-Release Defect Density (DD) Algorithm**

The post-release defect density algorithm consists of two steps given below:

**Step 1**: Collect the raw material: You are going to need the total no. of defects (for a release/build/cycle).

In this step, the number of defects in the development phase is calculated using an eclipse plugin known as Spot-bugs with version 3.1.12.r201903011242-190e1e at default analysis level for each open source system.

And the number of defects in the testing phase is calculated by counting issues raised by the users at the GitHub repository for the respective version of each open source system prior to the release.

**Step 2**: Calculate the average no. of defects/Functional area or KLOC

**Formula:**

**Defect Density = Defect count/size of the release (KLOC)**

The result shown in table represents the defect density for each open source system with respect to versions:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | **Version 1** | | | |
| **Defect during development** | **Defect found by user** | **KLOC** | **Defect Density** |
| ImageJ | 191 | 10 | 148.539 | 1.35 |
| Checksum | 16 | 2 | 2.783 | 6.47 |
| Checkstyle | 260 | 407 | 249.366 | 2.67 |

Table 5.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | **Version 2** | | | |
| **Defect during development** | **Defect found by user** | **KLOC** | **Defect Density** |
| ImageJ | 191 | 10 | 148.539 | 1.35 |
| Checksum | 16 | 2 | 2.799 | 6.45 |
| Checkstyle | 267 | 414 | 252.816 | 2.69 |

Table 5.2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | **Version 3** | | | |
| **Defect during development** | **Defect found by user** | **KLOC** | **Defect Density** |
| ImageJ | 181 | 11 | 149.039 | 1.28 |
| Checksum | 17 | 3 | 3.357 | 5.97 |
| Checkstyle | 267 | 424 | 254.287 | 2.71 |

Table 5.3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | **Version 4** | | | |
| **Defect during development** | **Defect found by user** | **KLOC** | **Defect Density** |
| ImageJ | 183 | 12 | 149.039 | 1.30 |
| Checksum | 16 | 3 | 3.570 | 5.32 |
| Checkstyle | 267 | 426 | 255.309 | 2.71 |

Table 5.4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | **Version 5** | | | |
| **Defect during development** | **Defect found by user** | **KLOC** | **Defect Density** |
| ImageJ | 183 | 12 | 149.039 | 1.30 |
| Checksum | 16 | 4 | 3.570 | 5.60 |
| Checkstyle | 267 | 445 | 255.309 | 2.78 |

Table 5.5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | **Version 6** | | | |
| **Defect during development** | **Defect found by user** | **KLOC** | **Defect Density** |
| ImageJ | 171 | 12 | 149.039 | 1.22 |
| Checksum | 15 | 5 | 4.255 | 4.7 |
| Checkstyle | 267 | 452 | 257.525 | 2.79 |

Table 5.6

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | **Version 7** | | | |
| **Defect during development** | **Defect found by user** | **KLOC** | **Defect Density** |
| ImageJ | 171 | 13 | 149.039 | 1.23 |
| Checksum | 15 | 5 | 4.821 | 4.14 |
| Checkstyle | 268 | 464 | 260.813 | 2.80 |

Table 5.7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | **Version 8** | | | |
| **Defect during development** | **Defect found by user** | **KLOC** | **Defect Density** |
| ImageJ | 171 | 13 | 149.041 | 1.23 |
| Checksum | 16 | 7 | 4.929 | 4.67 |
| Checkstyle | 268 | 482 | 260.898 | 2.87 |

Table 5.8

1. **Correlation Between Metric 5 and 6**

The following information can be inferred from the obtained results:

* + Defect removal efficiency is the ability of the development team to remove the defects prior to the release.
  + Post-release defect density is the number of unfound defects.
  + These two metrics are negatively correlated with each other.
  + System having highest defect removal efficiency should have least post-release defect density as during the development phase the developer’s or quality assurance groups keep the track of bugs they found and correct them before the release.
  + The ImageJ has the highest DRE and has lower DD. Moreover, DRE is lowest for the system checksum hence it has highest DD among other systems we have chosen.
  + The result validates the correlation between the metrics.
  + However, the post-release defect density (DD) also depends upon the size of the development team and the size of the system.

# **Correlation Between Metric 1, 2, and 6**

The following correlation can be depicted from the results

* + The increase in code coverage leads to a decrease in potential fault.
  + Branch and statement coverage are negatively correlated with Post-release defect density.
  + The test suites written for checksum are less effective as it has the lowest code coverage as compared to checkstyle and ImageJ hence, it has the highest DD.
  + The test suites with maximum coverage might reveal more defects, which in turn reduce the number of Post-release defect density assuming that the team is competent or efficient enough to remove the defects prior to the release (DRE).
  + Therefore, the higher the code coverage, the Lower will be the defect density after release.
  + The result validates the correlation between the metrics.

# **CONCLUSION**

We have selected the 3 open source system each with 8 versions and presented the analysis with respect to correlation between metrics. The metrics applied are strongly correlated to the object-oriented systems. At present, performed the analysis on limited versions of the system hence we are drawing the promising conclusion based on the results we have collected. The experiment performed on every selected version of chosen three open source systems, underline that code coverage and mutation score together effective to evaluate the effectiveness of the test suites. From our results, ImageJ has the highest code coverage and mutation score in

comparison with the other two systems. Hence, it has the greater effectiveness of their test suites to find the bugs in the source code. Developers also uses cyclomatic complexity to find the faulty areas in the software. From our research we found that there is a moderate correlation between the code coverage and cyclomatic complexity. We have also conducted the study to analyze the code coverage test cases and correlation between the post release defect density using the issue tracking system. We found that code coverage has insignificant correlation with finding the bugs after the release of the software. The test suites with maximum coverage might reveal defects, which in turn reduce the number of post release defect density assuming that the team is competent/ efficient enough to remove the defects prior to the release. Therefore, higher the coverage lower will be the defect release density. Metric 5 and 6 are negatively correlated with each other. Higher the development team efficient in detecting the bugs before the release of the software less is defect density of the system after the release.

# **Future Work**

In future, we would like to consider the following points to be covered in addition to this study:

* + Better issue tracking system to provide accuracy and effective result to find bugs by the user of each system as Github do not provide accurate data as an issue tracker.

1. **References**

[1] https://www.openhub.net/p/checkstyle

[2] https://www.openhub.net/p/image\_j

[3] https://www.openhub.net/p/checksum-maven-plugin

[4]https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=488361

[5] https://github.com/imagej/imagej1

[6] https://github.com/checkstyle/checkstyle

[7] https://github.com/nicoulaj/checksum-maven-plugin

[8] https://www.softwaretestinghelp.com/defect-density/

[9]http://www.ifpug.org/Documents/Toppin99percentDRE2016.pdf

[10]http://www.westfallteam.com/Papers/defect\_removal\_effectiveness.pdf

[11]http://www.divaportal.org/smash/get/diva2:826710/FULLTEXT01.pdf