Cover Letter

- Team

Name	Student ID	Email	Github Username
Anand Kacha	40047673	a_kacha@encs.concordia.ca	ndkcha
Shareen Ali	40075802	a_hareen@encs.concordia.ca	ShareenAli
Manpreet Singh	40083737	si_preet@encs.concordia.ca	manni1067
Shivani Jindal	40071296	shivani1995jindal@gmail.com	ShivaniJindal
Jasmeet Kaur	40088712	k_asmeet@encs.concordia.ca	Jasmeet8

- Replication Package

Github Url : https://github.com/ndkcha/measurement-metrics

Software Measurement Metrics Report

Anand Kacha
Master of Applied Computer Science
Concordia University
Montreal, Canada
a_kacha@encs.concordia.ca

Shareen Ali
Master of Software Engineering
Concordia University
Montreal, Canada
a_hareen@encs.concordia.ca

Manpreet Singh
Master of Software Engineering
Concordia University
Montreal, Canada
si_preet@encs.concordia.ca

Jasmeet Kaur
Master of Software Engineering
Concordia University
Montreal, Canada
k_asmeet@encs.concordia.ca

Shivani Jindal
Master of Software Engineering
Concordia University
Montreal, Canada
shivani1995jindal@gmail.com

***Abstract*—Software measurement is one of the most important parts in the Software Industry. The software is measured to ensure that it is better in every aspect including development, quality and maintenance. In order to determine the importance of the measurement techniques in Software Engineering, we have measured five different metrics on five different open source projects, and provided the correlation analysis on as much metrics as possible for each project.**

Keywords—software, engineering, measurement, metrics, code coverage, test suit effectiveness, complexity.

I. Introduction

Every organization in Software Industry aim to improve their software in all the aspects. It can be from the design process to the post release maintenance. All the companies make an effort to achieve such a target with as minimum resource as possible with respect to time and cost. In order to accomplish the goal, the companies require the better estimation of such parameters. We can calculate many measurements by taking different parameters in consideration, and determine the appropriate numeric values to these measurements. Such methods are called Software Metrics, and they provide formal means of tools to estimate the internal and external measurements of the software.

The industry uses the Software Metrics in order to improve the decision making during the design phase as well as the development phase of the software. Different metrics may have a different role for the measurement of different kind of Software. With only useful metrics of the software, we can achieve the best results in effectiveness and efficiency. Since the same metric may behave differently in different kind and volume of the software, sometimes it may be trivial to determine the perfect metric to measure the software.

With Software Metric, the managers in the software development team often get the clear idea of return on investment (RoI), different aspects for improvement, manage the team with respect to workload, time management and overall cost. Calculating different software metrics helps the entire team in their own ways. For example, the managers can benefit from the metrics to prioritize and track the issues such a way that they can improve the productivity of the team. It might lead to more enhancement management and more quality software product. Just like the managers, developers can also benefit from the measurement metrics. For example, the development team often use the code coverage or product related metrics to communicate the status/quality of the project. The calculation of the metrics often has multiple interpretation and/or definitions and/or calculation techniques. It might lead to inconsistency and lack of understanding of the metric for its purpose. The metrics can be computed more accurately if they are calculated by the development team. The metrics must have several important characteristics, such as, Simple and computable, Consistent and unambiguous (objective), Use consistent units of measurement, Independent of programming languages, Easy to calibrate and adaptable, Easy and cost-effective to obtain, Able to be validated for accuracy and reliability, and Relevant to the development of high-quality software products.

Here is a list of metrics that we will discuss further in this paper:

1. Test Coverage Metrics: Statement and Branch Coverage
2. Test Suit Effectiveness: Mutation Score
3. Complexity Metric: McCabe Complexity by calculating Cyclomatic Complexity
4. Software Maintenance Metric: Adaptive Maintenance Effort Model
5. Software Quality Metric: Post-release Defect Density

In this paper, we will also perform the correlation analysis amongst different kind of metrics. The correlation coefficient is a measure of linear association between two variables (here in our case, it will be the values of metrics). We have calculated the Spearman as well as Pearson correlation to find the correlation amongst these metrics.

II. Test Subjects

A. *Openfire*

Size: 297k LOC
Version Control System: Git
Issue Tracking System: Jira
(project) - https://github.com/igniterealtime/Openfire
(bug tracking) - https://issues.igniterealtime.org/projects/OF/issues/OF-870?filter=allopenissues

Openfire is a real time collaboration (RTC) server licensed under the Open Source Apache License. It uses the only widely adopted open protocol for instant messaging, XMPP

(also called Jabber). Openfire is incredibly easy to setup and administer, but offers rock-solid security and performance.

Openfire is a XMPP server licensed under the Open Source Apache License.

B. *Commons IO*

Size: 66k LOC

Version Control System: Git

Issue Tracking System: Jira

(project) - https://github.com/apache/commons-io

(bug tracking) - https://issues.apache.org/jira/projects/IO/issues/IO-552?filter=allopenissues

The Apache Commons IO library contains utility classes, stream implementations, file filters, file comparators, endian transformation classes, and much more.

The Apache Commons IO library is under the Apache Licence v2.

C. *Commons Collections*

Size: 126k LOC

Version Control System: Git

Issue Tracking System: Jira

(project) - https://github.com/apache/commons-collections

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

The Java Collections Framework was a major addition in JDK 1.2. It added many powerful data structures that accelerate development of most significant Java applications. Since that time it has become the recognised standard for collection handling in Java. Commons-Collections seek to build upon the JDK classes by providing new interfaces, implementations and utilities.

The Apache Commons Collection library is under the Apache Licence v2.

D. *jFreeChart*

Size: 297k LOC

Version Control System: Git

Issue Tracking System: Github

(project) - https://github.com/jfree/jfreechart

(bug tracking) - https://github.com/jfree/jfreechart/issues

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).

The library is licensed under the terms of the GNU Lesser General Public License (LGPL) version 2.1 or later.

E. *jSoup*

Size: 38k LOC

Version Control System: Git

Issue Tracking System: Github

(project) - https://github.com/jhy/jsoup

(bug tracking) - https://github.com/jhy/jsoup/issues

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).

jsoup is an open source project distributed under the liberal MIT license.

III. METRICS

The software measurement metrics generally helps the team to understand the different aspect of the system. It follows no standard values or definitions to the development team. It often has different value to different teams. It solely depends on the kind of goal the team is trying to achieve.

Here are the five different metrics that we considered to measure the five test subjects mentioned above:

A. *Test Coverage Metrics*

Code inclusion measurements, to a great extent got from measurements utilized in programming testing, distinguish which structure classes in the equipment depiction language (HDL) code to practice during reenactment. Thus, likewise called Test inclusion measurements.

Progressively refined code-based measurements utilized in equipment confirmation are branch, articulation, and way inclusion. These measurements include the control course through the HDL code during recreation and are best depicted by the control stream chart (CFG) relating to that code. Control explanations (the announcements from which control can hop to one of a few spots) comprise the expanding focuses in the CFG—for instance, if, case, or while articulations.

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 measurement, which is utilized to compute and quantify the quantity of explanations in the source code which have been executed. Utilizing this system, we can check what the source code is required to do and what it ought not. It can likewise be utilized to check the nature of the code and the progression of various ways in the program. It is otherwise called Segment Coverage or Line Coverage.

Statement coverage = (Number of statements executed / Total number of statements) *100%

One of the advantages of using this metric can be the ability to check the flow of statements executed. It covers the statements that are executed in different scenarios. If in any case, the statement is missed in the execution cycle, the development team can figure out the issue associated with it and would solve it before it becomes the major issue in the production environment. Another advantage is that it confirms the code quality. The better code is one that has maximum code coverage. It means that the most of the statements written to the develop the software are executed perfectly. It helps developer improve the development techniques and algorithmic design of the module they are working on.

One downfall of this metric can be the inability to detect the possible infinite cycles in the loop. While this metric can only determine if the statement is executed or not, it cannot determine if a loop is terminated or not. This metric only covers the true conditions in order to measure the quality of the code which may not be the best idea.

The best part of this testing technique is that it can be conducted by the code developers themselves.

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. That is, every branch taken each way, true and false. It is also known as All-edge coverage or Decision coverage.

Branch coverage = (Number of decision outcomes tested / Total Number of decision Outcomes) * 100%

This metric helps to validate that all the branches in the code are reachable in the execution environment. It makes sure that no branches lead to any kind of unexpected behavior during the execution of the system. In conclusion, it can eliminate the problem we faced in the statement coverage.

This metric ignores the branches within Boolean expressions which are the outcome of short-circuit operations. We have to take note that 100% branch coverage guarantees 100% statement coverage, but 100% statement coverage does not guarantee 100% branch coverage.

Collecting Data

We can collect the data for both Branch and Statement coverage using JaCoCo tool. To execute this tool, we need to run the test phase of the test subjects and the results are generated.

We have to follow certain steps in order to compute the results from JaCoCo as follows:

- Add the JaCoCo plugin in the build section in the pom.xml of the project
- Configure the JaCoCo plugin to run during the test phase and specify the output directories
- Run the clean task if necessary, to wipe out unnecessary build files
- Run the test task in the maven builder
- Results are generated in the target/reports directory

Since we have also used the IntelliJ IDEA to open the projects into and run the tasks, we can use the IDE's built-in facilities to calculate the code coverage. We can configure JaCoCo in the Code Coverage section of the build configurations in IntelliJ. Here we have to take a note that we should modify only the test task's build configuration in order to use JaCoCo for Coverage. And then when we run the test task with coverage option (right click on task and choose Run 'test' with coverage), it will generate the coverage reports using JaCoCo automatically.

The coverage reports are exported in HTML, XML or CSV formats. We can use different formats for different purposes. The HTML format will give us the visual idea of how well the code is covered. But full coverage does not always mean effective testing. It can be interpreted as the amount of code executed during the (unit) tests.

Apache Commons Collections

Apache Commons Collections

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Miss
org.apache.commons.collections4.map		88%		78%	364	1,716	289	3,194	
org.apache.commons.collections4.trie		77%		71%	159	509	202	928	
org.apache.commons.collections4		88%		86%	197	1,081	189	1,641	
org.apache.commons.collections4.iterators		84%		86%	124	608	180	1,149	
org.apache.commons.collections4.bidimap		90%		88%	86	577	105	1,171	
org.apache.commons.collections4.multimap		77%		67%	81	291	121	506	
org.apache.commons.collections4.multiset		72%		60%	68	219	117	418	
org.apache.commons.collections4.functors		85%		85%	99	455	104	761	
org.apache.commons.collections4.list		93%		89%	72	568	85	1,218	
org.apache.commons.collections4.bag		87%		87%	41	255	55	484	
org.apache.commons.collections4.set		88%		87%	42	285	51	465	
org.apache.commons.collections4.comparators		82%		69%	60	159	35	228	
org.apache.commons.collections4.collection		89%		87%	30	212	45	405	
org.apache.commons.collections4.queue		91%		93%	14	113	21	211	
org.apache.commons.collections4.sequence		90%		92%	10	73	19	141	
org.apache.commons.collections4.splitmap		77%		58%	10	33	12	53	
org.apache.commons.collections4.keyvalue		97%		84%	14	96	5	141	
org.apache.commons.collections4.trie.analyzer		92%		71%	11	25	4	40	
org.apache.commons.collections4.properties		100%		100%	0	3	0	8	
Total	7,392 of 54,672	86%	1,155 of 6,189	81%	1,482	7,278	1,639	13,162	5

[Figure 1] JaCoCo Report for the entire project

B. Test Suit Effectiveness (Mutation Testing)

Software testing is an important part of the software engineering lifecycle process. It plays a pivotal role in the quality assurance of the software product. During software maintenance, testing is a crucial activity to ensure the quality of program code as it evolves over time. With the increasing size and complexity of software, adequate software testing has become increasingly important. In order to measure the effectiveness, the quality of test suites developed is measured. Good software testing has software tests than detect and find faults, if a test cases is unable to detect fault it will be hard to build a good software. This is fulfilled by Mutation Testing.

It's a kind of Whitebox-testing used to develop the hidden test cases by modifying code and check if it could detect the error probability. Furthermore, changes in code statements are made at the lower level so that the meaning of the code remains neutral. The changed statements are the called the mutant and the score derived is known as the mutation score. Thus, to know the failed mutant codes, the statement should be vigorous.

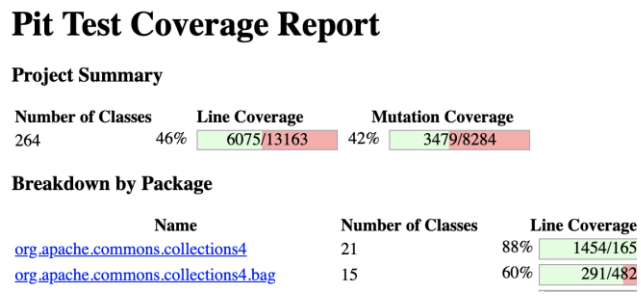
Mutation Score can be defined as **the number of Mutants that were killed divided by the total number of Mutants multiplied by 100.**

Collecting Data

We can collect the data for mutation testing using Pit-Test tool. This tool can be installed using Maven/Gradle builder. We only have to add the correct version and configuration for it in the builder's root file. To execute this tool, we need to run the test phase of the test subjects and the results are generated.

We have to follow certain steps in order to compute the results from Pit-Test as follows:

- Add the Pit-Test plugin in the build section in the pom.xml of the project
- Configure the Pit-Test plugin to run during the test phase (specify the goal as mutationCoverage)
- Run the clean task if necessary, to wipe out unnecessary build files
- Run the test task in the maven builder
- Results are generated in the target/pit-reports directory



Pit Test Coverage Report

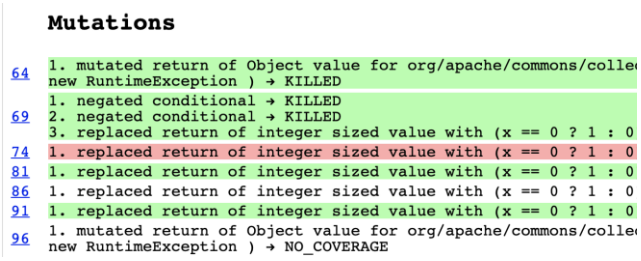
Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
264	46%	6075/13163	42%	3479/8284

Breakdown by Package

Name	Number of Classes	Line Coverage	
org.apache.commons.collections4	21	88%	1454/165
org.apache.commons.collections4.bag	15	60%	291/482

[Figure 2] Pit Test Coverage Report of the Project



Mutations

```
64  1. mutated return of Object value for org/apache/commons/colle
    new RuntimeException ) → KILLED
69  1. negated conditional → KILLED
    2. negated conditional → KILLED
    3. replaced return of integer sized value with (x == 0 ? 1 : 0
74  1. replaced return of integer sized value with (x == 0 ? 1 : 0
81  1. replaced return of integer sized value with (x == 0 ? 1 : 0
86  1. replaced return of integer sized value with (x == 0 ? 1 : 0
91  1. replaced return of integer sized value with (x == 0 ? 1 : 0
96  1. mutated return of Object value for org/apache/commons/colle
    new RuntimeException ) → NO_COVERAGE
```

[Figure 3] Mutations performed during the Pit Test

C. Complexity Metric (McCabe Complexity)

In Software Development, it is very difficult to introduce more features or maintain the complex code. There are two fundamental approaches to identify the complexity. One of them calculates the essential complexity and another calculates the accidental complexity. Essential complexity is inevitable. The business logic can be one of the factors to this complexity. Accidental complexity is generally the byproduct of the rush into the development. It can be in any form, such as quick fixes, in-experienced developers, too much workload in one sprint so the developers would do anything to make the code work, etc. The McCabe complexity metric uses cyclomatic complexity to measure the complexity. It is one of the approaches to calculate the accidental complexity.

Cyclomatic Complexity is a mathematical technique to modularize the program. Thomas McCabe suggested that every possible control flow graph has a finite number of independent paths. He derived the formula to measure the complexity metric as,

CC = E - N + 2P or CC = D + 1

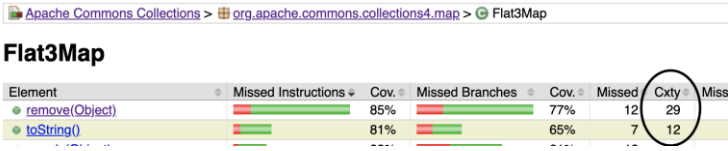
Where,

- E is the number of edges in the graph
- N is the number of nodes in the graph
- P is the number of connected components in the graph
- D is the control predicates in the graph

The cyclomatic complexity calculate the total possible linearly independent paths that are being executed inside the method (or any other component). And out of all the paths, it calculates the complexity based on the execution flow of method or the component. The more branch it has with a smaller number of nodes, the more complex the code becomes. In other words, very few control predicates (such as, if, for, while, etc.) will keep the complexity to a lower value. The paths determined to help calculate the complexity also help in determining the number of test cases the testing team needs to write in order to cover the entire method (or the component).

This approach of calculating the complexity has its own pitfalls. One of the fundamental disadvantages is that the complexity metric sees all kinds of control predicates equally complex. For example, a loop executes a statement multiple times and it also adds more the complexity when the developers code it. While an if condition executes a statement only once and it adds relevantly less complexity with respect to a loop when the developers code it. But the cyclomatic complexity considers both the same, which may not be the best way to determine the complexity. The complexity also does not take testing to consideration. When we nest the loop, the algorithmic complexity is o(n^2) which is significantly larger than the single loop (o(n)). But the cyclomatic complexity sees both as a same.

According to the JaCoCo documentation, "JaCoCo also calculates cyclomatic complexity for each non-abstract method and summarizes complexity for classes, packages and groups". We used the same configuration as we had for the coverage metrics to calculate the complexity. In the JaCoCo report, we can see the cyclomatic complexity field along with the other coverage metrics.



Apache Commons Collections > org.apache.commons.collections4.map > Flat3Map

Flat3Map

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Miss
remove(Object)		85%		77%	12	29	
toString()		81%		65%	7	12	

[Figure 4] Cyclomatic complexity in JaCoCo report

D. Software Maintenance Metric (Adaptive Maintenance Effort Model)

In Software Development, at the point when the product needs to experience the adjustment in the framework, the group frequently belittles the time and exertion required to roll out the improvements. This likewise prompts an absence of approval, acknowledgment, arranging, evaluating, and support in the product upkeep cycle. To process this measurement, we determine the estimation model utilizing relapse examination on the information that we gathered. At that point we utilize the forget one way to deal with approve the model.

For the calculations, we can treat the change in the LOC as a parameter to the regression model. The DLOC we can determine from the project's version control system based on the diff in versions of the file. We then can determine the data points from the available data. Once we determine the DLOC, we can move forward to calculate the maintenance effort using the following formula:

E = -40 + 6.56DLOC

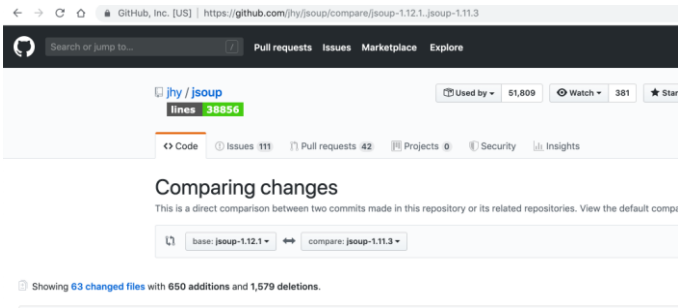
Where DLOC is the difference in lines of code.

The adaptive maintenance effort can also be calculated for the regression analysis. Such analysis can also be useful to train the machine learning model in order to predict the future efforts for given requirements. The regression analysis on the metric can also help us understand the relationship between the variables used and the effort during the maintenance process.

The only pitfall we can conclude is that the outcome of this metric, when used for machine learning, it is split between the training and testing data. And such partition may disturb the

distribution of the data which may lead to inconsistent results every time we run the algorithm.

To calculate this metric, the version control system provides the functionality to access the changes between two different versions of the project with url https://github.com/<path-to-project>/compare/version1..version2. We can visually determine the changes in the code between two versions as follows,



[Figure 5] Visualize the changes between two versions of the project

For the analysis, we need these results in the form we can perform our calculations via code or excel tools. Such functionality is provided by the git protocol with the command as following:

git log version1..version2 --pretty=format:\"%an\" --shortstat #{ARGV.join(' ')}

The above command gives us the list of changes made to the project between two versions with respect to the authors as well as the overall project in CSV format which we can later use in Microsoft Excel for further calculations.

	A	B	C	D	E
1	Henri Yandell	5073		Total DLOC:	6225
2	Niall Kegan Pemberton	555		Total Effort:	40796
3	Stephen Colebourne	424			
4	Sebastian Bazley	90			
5	Rahul Akolkar	42			
6	Matthew Jason Benson	28			
7	Emmanuel Bourg	9			
8	Dennis Lundberg	4			
9					

[Table 1] Snapshot of excel sheet demonstrating the portion of the calculation

The detailed scripts and the gathered data are packed along with the replication package.

E. Software QualityMetric (Post Release Defect Density)

Defect is what is the deviation observed from an expected behavior. Once the software is released to the customer and then if defects are encountered, is referred as post-release defects. Defect density is the total count of defects found per 1000 lines of source code in a software after software has been released. Defect density is used as the indicator for the Product Quality.

Defect Density (DD) = Defect Count (Dc) / Size of the Software (SS)

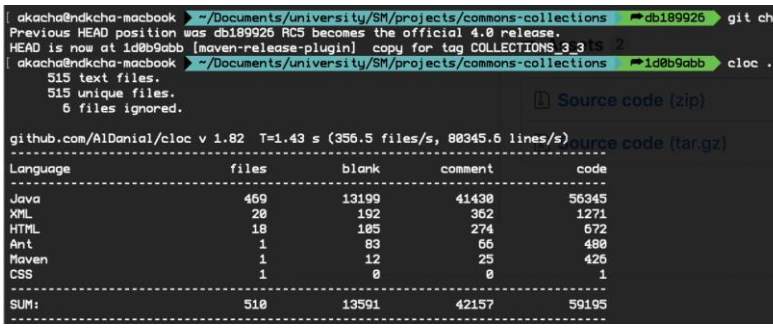
The defect density can be affected by the type of issues that we consider from the issue tracking system for the calculation. We only use the issues detected by the in-house team of the project (which they track using Jira or other issue tracking system). This metric also depends on the quality of the testing team as well.

There are multiple kinds of issue tracked using JIRA or other tracking systems. One of them being the stories, which are not bugs, but the features or pre-release issues that developers usually work on. Another being the tasks, which are part of the stories. Each individual developer is assigned a task to work on. Another kind of issue is bugs. They are generally detected when the stories are deployed to the production environment and the testing team are testing for the runtime defects in the system. If any anomaly is found, it is tracked a bug.

We also need the source lines of code to calculate the size of the system. There are many tools available to gather this data. One such tool that we use is cloc (https://github.com/AlDanial/cloc/). It can be installed using the following command on Mac Terminal:

brew install cloc

The data collection using this tool is illustrated as below:



[Figure 6] Calculating the source lines of code

We also need the number of bugs to calculate this metric. We can calculate the number of issues for different versions on Jira from the user interface and then export them as a CSV from the user interface itself. In order to achieve so, we first have to filter the issues on JIRA by the type "Bug" (the one with red icon). And we can apply another filter with the version number and then we can see the list. With the option on the top right corner, we can export the list to the CSV for further calculations.

Data collected from the JIRA

Name	Defect Count	Size (in K)	Defect Density
Commons Collections	17	127	0.13385827
Commons IO	50	66	0.75757576
Openfire	117	296	0.39527027
jFreeChart	49	297	0.16498316
jSoup	111	38	2.92105263

[Table 2] Post Release Defect Density

As we can see from the table above, the number of bugs is not always propositional to the size of code. It can be propositional to the size between two versions in the same project. But we cannot generalize that assumption. Hence, we can conclude that the number of bugs in the system does not solely depend on the size of the project. There are other factors

as well. One of the factors can be the complexity as well as the quality of the development team.

IV. Correlation Analysis

Once we gather the data for all the metrics, we can calculate the correlation amongst them in order to determine how they are related.

A. *Correlation between each coverage metric and test suits effectiveness*

The rationale behind this correlation is that the test suits with higher coverage might show better suite effectiveness.

The collected data are stored in the data directory of the replication package. Since the HTML reports helps visualize the data better, they are also included in the directory. The shown below show the overall coverage as mentioned in the report file. But the correlation calculated are based on results generated from the individual classes.

Name	Statement Coverage	Mutation Testing	Correlation (Spearman)	Correlation (Pearson)
Commons Collections	86%	42%	0.30396665	0.392109
Commons IO	90%	80%	0.89260428	0.613601
Openfire	7%	7%	0.87717638	0.868728
jFreeChart	58%	33%	0.86397328	0.844245
jSoup	83%	68%	0.9171613	0.441793

[Table 3] Correlation between statement coverage and mutation testing

From the table, we can see that the despite of having poor coverage in Openfire project the correlation between them comes on strong. That is because we calculate the correlations with respect to data collected from the classes. In the openfire, not all the classes are tested, but off all the classes are tested, most of them passes the mutation testing criteria. Hence, there is a strong correlation between them. And Mutation Score does not strongly increase with the coverage for commons collections project. This shows that sometimes Code coverage is moderately correlated to Mutation score for the particular project. Other than that, the analysis show that the test suits with higher coverage might show better suite effectiveness for most of the projects. There is an exception that we have in our test subjects that show that the test suits effectiveness does not solely depend on the test coverage. There are other factors associated with the test suit effectiveness such as quality of the tests suits.

The analysis for the branch coverage is also similar to the statement coverage. It is also included in the replication package along with this paper.

B. *Correlation between each coverage metric and complexity metric.*

The rationale behind this correlation is that classes with higher complexity are less likely to have high coverage test suites.

In order to calculate the correlation between these metrics, we used the JaCoCo tool to calculate each metric.

Name	Branch Coverage	Cyclomatic Complexity	Correlation (Spearman)	Correlation (Pearson)
Commons Collections	81%	8	0.06159276	0.091486
Commons IO	80%	12	0.28571429	0.311865
Openfire	6%	12	0.85003806	0.243533
jFreeChart	46%	18	0.61062684	0.328758
jSoup	78%	10	0.39931084	0.305696

[Table 4] Correlation between branch coverage and cyclomatic complexity

From the table, we can see that the Commons Collections and jSoup have relatively high coverage and their complexity are relatively low. And openfire and jFreeChart have relatively low coverage and their complexity is relatively high. But the Commons IO suggests that despite the coverage being high, the complexity is not relavently low. Such exceptional result in our test subjects suggest that the cyclomatic complexity may not consider the flow of data for the particular component.

C. *Correlation between each coverage metric and software quality metric.*

The rationale behind this correlation is that large systems with low test coverage contain relatively more bugs. But there are other factors also that can affect the relationship between these two metrics.

For the below table, we considered the latest state of the system in issue tracking system as well as the code.

Name	Statement Coverage	Branch Coverage	Bugs
Commons Collections	86%	81%	17
Commons IO	90%	80%	50
Openfire	7%	6%	117
jFreeChart	58%	46%	49
jSoup	83%	78%	111

[Table 5] Correlation between Defects and Coverage Metrics

From the table, the bug distribution of the bugs to the coverage help us conclude that the less covered code indeed has more bugs and vice versa. But the number of bugs also depend on several other factors, such as quality of the build, quality of the development team, experience of the testing team, the tools used for testing, etc. Our results conclude that the Spearman correlation between statement coverage and the post release defect density is 0.1, Pearson correlation is 0.353586607, and Pearson correlation between branch coverage and the post release defect density is -0.2, Pearson correlation is 0.35823387.

D. *Correlation between software quality metric and software maintenance effort.*

The rationale behind this correlation is that the more effort we make to maintain the software, the less defects we can encounter after the release.

	Bugs	Effort
Openfire	0.39527027	166542.016
Commons IO	0.757575758	43955.296
Commons Collections	0.133858268	264237.472
jFreeChart	0.164983165	110581.28
jSoup	2.921052632	8909.152

[Table 6] Results for Post release defect density and Maintenance Effort

From the table, we can see that the more effort the team makes in maintaining the software the less bugs we can encounter after the release. In the table, the column bugs represent the defect density. We can strongly suggest that maintain the software system is an important aspect in order to improve the system with respect to bugs and many other aspects. We achieved the strong Spearman correlation of –0.9 and the Pearson correlation of -0.717431289 during our analysis which shows the importance of the maintenance efforts in the software development.

Observations on Correlations

- While higher test coverage might have the better test suite effectiveness for most of the projects, but it cannot be generalized to all the systems and teams. It also depends on the quality of the test suits and experience of the developers in automated testing.
- Cyclomatic complexity helps determine the complexity of the system with respect to the coverage of the system. If we are able to test more components in the system, the code becomes less complex.
- Even though the correlation between cyclomatic complexity and coverage metrics is better than the test suite effectiveness and coverage metrics. Both the metrics cannot be under-estimated or over-estimated. They have an important role to play in their own respective ways.
- Quality of the software depends on the quality of the entire team. It does not just depend on the coverage of the tests or the quality of development team.
- In large project, if the testing is not part of the project from the beginning, it is harder to identify the bugs from the development teams and the post release defects may increase exponentially. Even though they start testing the code towards the end of development, it is difficult to cover the entire code before the release and measure the software automatically. Such large projects require more time into production; hence it pushes the deadline and the delivery is not efficient.
- After the software is deployed, it is a crucial to maintain the software in order to improve the quality of the software. The maintenance of the software often leads to less bugs in the production environment.

V. Tools And Calculations

A. *JaCoCo*

The JaCoCo is developed by EclEmma team. It is a popular tool to generate the coverage reports of the Java project. It is available for most of the build tools used in Java (Maven, Gradle, etc.). It is also available for the IntelliJ IDEA IDE as a plugin. We can enable this plugin and use it by configuring the run configurations of the tasks in IntelliJ. It uses many automated testing frameworks when calculating the code coverage such as jUnit, etc. And it supports the large number of versions of these testing frameworks as well. The configuration steps are mentioned along with the replication package.

B. *CLOC*

We have used cloc tool to calculate the Source Lines of Code. It is developed by AlDanial and hosted on github. We can download this tool in all the operating systems. Since we are using MacOS Mojave, we installed it using brew package manager in darwin environment using the following command:

brew install cloc

We can use this tool using the command line interface of the operating system. All we have to do is go to the root directory of the project and run “cloc .” command and it will calculate the SLOC for us.

This tool supports many programming languages. The detailed guideline to use this tool is available here (https://github.com/AlDanial/cloc/)

C. *PIT-Testing tool*

We have used PIT test tool for calculating mutation coverage in our project. PIT is a state-of-the-art mutation testing system, providing gold standard test coverage for Java and the jvm. It's fast, scalable and integrates with modern test and build tooling. The reports produced by PIT are in an easy to read format combining line coverage and mutation coverage information.

We can configure the PIT test tool as a plugin to the maven or gradle repositories. Once we add the maven dependency of PIT test tool, the IntelliJ IDEA automatically imports the plugin binaries to make it run with the test task of the project.

D. *Correlations (Microsoft Excel)*

We have used Microsoft Excel to perform the calculation on the data that we have gathered using the above tools. Microsoft Excel has built-in functions to calculate the correlation between two data sets. We can use the RANK.AVG function to calculate the rank and CORREL function to calculate the correlations. The detailed configuration steps are mentioned along with the replication package.

VI. Related Work

- According to paper, “Coverage Is Not Strongly Correlated with Test Suite Effectiveness” who tests on Java projects there is a low to moderate correlation between coverage and effectiveness when the number of test cases in the suite is controlled for. In addition, they found that stronger forms of coverage do not provide greater insight into the effectiveness of the suite. Their results suggest that coverage, while useful for identifying under-tested parts of a program, should not be used as a quality target because it is not a good indicator of test suite effectiveness.

- For most reasonable definitions of coverage metrics, the effort invested in deriving the metrics and measuring them pays off in the form of better error detection. The value of increased confidence in the design's correctness almost always outweighs the overhead of measuring coverage. Because this confidence depends on the connection between bug classes and coverage metrics.
- According to paper, "Coverage Is Not Strongly Correlated with Test Suite Effectiveness" The coverage of a test suite is often used as a proxy for its ability to detect faults. However, previous studies that investigated the correlation between code coverage and test suite effectiveness have failed to reach a consensus about the nature and strength of the relationship between these test suite characteristics. Moreover, many of the studies were done with small or synthetic programs, making it unclear whether their results generalize to larger programs, and some of the studies did not account for the confounding influence of test suite size.
- The cyclomatic complexity is the commonly used metric in software engineering. It is based on the control flow structure of the program. Each procedure's statements are considered as a graph. And the complexity is measured by linearly independent paths inside the graph.
- There has been a significant number of effort estimation models in practice. The commonly referred is the Constructive Cost Model, which supports the cost estimation, effort, and timeline for the project. Some model suggests the use of estimated cost or/and the algorithmic cost and/or the function points as we covered in the lecture. The relevant papers related to such work cited in the references.
- With the growing era of Machine Learning, there are recently surfaced models that could be used to measure the efforts in Software Maintenance. Gray and MacDonell used ML-based algorithms such as Artificial Neural Network, Regression Analysis, and a measurement model based on function point, and also the fuzzy logic. With all their analysis, they concluded that the model that uses fuzzy logic proved to perform well with various inputs, but it gave a bit poor accuracy compared to the Artificial Neural Network. And the research has shown that the stepwise regression with bi-directional elimination method provides better accuracy than the linear regression.
- According to the research work, "Estimate the post-release Defect Density based on the Test Level Quality", the author had a focus on today's software industry, that concentrates on enhancing the quality of a software product, which can be done during testing and development phase by improving the code quality. But to measure the effectiveness of code, post release defect density is considered an important metric that can be calculated by author "as the number of detected defects per 1000 lines of source code." Then it was concluded that the post release defect density of a software is decreased to half of its value, when test level quality is increased from lower level. Thus, they follow an inverse relation and hence, defect density is evaluated as an important factor in determining intrinsic product quality.

VII. Summary and Conclusion

We have used five test subjects to calculate the correlations amongst the measurement metrics. Based on the observation from the results, we conclude that a single metric or a specific metric is not suitable to give the accurate estimate of the system. These techniques of the measurement provide the overall idea about the system. We also need to take an account of internal product measurement as well as the measurements based on the nature and experience of the team. However, the metrics calculated using the software measurement techniques are also reliable in their own respective fields.

VIII. References

[1] Laura Inozemtseva and Reid Holmes, School of Computer Science University of Waterloo Waterloo, ON, Canada. Coverage Is Not Strongly Correlated with Test Suite Effectiveness http://www.linozemtseva.com/research/2014/icse/coverage/coverage_paper.pdf

[2] Serdar Tasiran, Compaq Systems Research Center Kurt Keutzer University of California, Berkeley. Coverage Metrics for Functional Validation of Hardware Designs https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=936247&tag=1

[3] Thomas J McCabe. A Complexity Measure. I", IEEE Transactions on Software Engineering.

[4] Ayman Madi, Oussama Kassem Zein and Seifedine Kadry. "On the Improvement of Cyclomatic Complexity Metric". International Journal of Software Engineering and Its Applications

[5] Jane Huffman Hayes, Sandip C Patel, Liming Zhaom, "A Metrics-Based Software Maintenance Effort Model". Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.

[6] Vinke, L. (2011). Estimate the post-release Defect Density based on the Test Level Quality. Retrieved February 09, 2019, from https://research.infosupport.com/wp-content/uploads/2017/08/MasterThesis-LammertVinke-Final.pdf.

[7] What Are Software Metrics and How Can You Track Them? https://stackify.com/track-software-metrics/

[8] Importance Of Software Measurement And Metrics. https://www.ukessays.com/essays/information-technology/importance-of-software-measurement-and-metrics-information-technology-essay.php

[9] JaCoCo Documentation (https://www.eclemma.org/jacoco/trunk/index.html)

[10] PIT Test http://pitest.org/

[11] Project Proposal of Team K