Cover Letter

- **Team I**

Name	Student ID	Email	Github Username
Iknoor Singh Arora	40082312	iknoorcan123@gmail.com	Iknoor99
Sukhpreet Singh Bhatia	40083564	sukhpreetbhatia025@gmail.com	Ssdev27
Aditi Bhayana	40083419	aditibhayana029@gmail.com	Aditi2904
Aakash Ahuja	40082822	aakashahuja1993@gmail.com	ahujakash
Ashmeet Singh	40070369	ashu6811singh@gmail.com	ashu6811

- Replication Package

Github Url : https://github.com/ssbDev27/MeasurementProject

Software Measurement Metrics Report

Aditi Bhayana
Master of Applied Computer Science
Concordia University
Montreal, Canada
aditibhayana29@gamil.com

Aakash Ahuja
Master of Software Engineering
Concordia University
Montreal, Canada
aakashahuja1993@gmail.com

Iknoor Singh Arora
Master of Applied Computer Science
Concordia University
Montreal, Canada
iknoorcan123@gmail.com

Ashmeet Singh
Master of Software Engineering
Concordia University
Montreal, Canada
ashmeet.singh6811@gmail.com

Sukhpreet Singh Bhatia
Master of Applied Computer Science
Concordia University
Montreal, Canada
sukhpreetbhatia025@gmail.com

***Abstract*—Software measurement is a quantified attribute of a characteristic of a software product or the software process .It is one of the most important parts in the Software Industry. The software is measured to anticipate future qualities of the product or process to enhance and maintain the quality of a product or process. In order to determine the importance of the measurement techniques in Software Engineering, we have measured five different metrics on four different open source projects, and provided the correlation analysis on as many metrics as possible for each project.**

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

I. INTRODUCTION

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.

As the software evolves, the code and the architecture become more complex, which can inevitably induce bugs in the system. Finding bugs later in the developmental stages can increase the cost of software remodeling.

Software developers usually use various metrics for determining the quality of the system. Using these metrics for predicting design defects during development as well as every stage can

increase the efficiency of the overall development process.

With Software Metric, the managers in the software development team often get the clear idea of return on investment, different aspects of improvement; to plan about the productivity and effort of the resources, schedule 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 can lead to delivery of quality software products. 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. In Software Maintenance, changes made on a system after its delivery takes up to 90% of all total cost of a typical software. Introducing new functionalities, altering bugs, and changing the code to improve its quality are major parts. There has been much research concentrating on the study of bad design practices, also called smells, defects, ant patterns or anomalies in the literature 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:Test Coverage Metrics: Statement and Branch Coverage

- Test Suit Effectiveness: Mutation Score
- Complexity Metric: McCabe Complexity by calculating Cyclomatic Complexity
- SoftwareMaintenanceMetric:Adaptive Maintenance Effort Model
- Software Quality Metric: Post-release Defect Density

In this paper, we will also perform correlation analysis amongst different kinds 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. Apache Commons Net

Size: 63k LOC

Version Control System: Git

Issue Tracking System: Jira

Project-https://github.com/apache/commons-net

BugTracking-https://issues.apache.org/jira/projects/NET/issues/NET-679?filter=allopenissues

Apache Commons Net library implements the client side of many basic Internet protocols. The purpose of the library is to provide fundamental protocol access, not higher-level abstractions. Apache Commons Net library contains a collection of network utilities and protocol implementations. Supported protocols include: Echo, Finger, FTP, NNTP, NTP, POP3(S), SMTP(S), Telnet, Whois

B. JFreeChart

Size: 297k LOC
Version Control System: Git
Issue Tracking System: Github
Project - https://github.com/jfree/jfreechart
BugTracking-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.

C. Commons Collections

Size: 127k LOC
Version Control System: Git
Issue Tracking System: Jira
Project:https://github.com/apache/commonscollections
BugTracking:https://issues.apache.org/jira/projects/COLLECTIONS/issues/COLLECTIONS714?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. Commons maths

Size: 388k LOC
Version Control System: Git
Issue Tracking System: Jira
Project-https://github.com/apache/commons-math
BugTracking-https://issues.apache.org/jira/browse/MATH-1529?jql=project%20%3D%20MATH%20ORDER%20BY%20key%20DESC

Commons Math is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language or Commons Lang.

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

III. METRICS

A software metric is a measure of software characteristics which are quantifiable or countable. Software metrics are important for many reasons, including measuring software performance, planning work items, measuring productivity, and many other uses.

Benefits of software metrics:

- Increase return on investment (ROI) Software development
- Identify areas of improvement
- Manage workloads
- Reduce overtime
- Reduce costs

A - Statement Coverage:

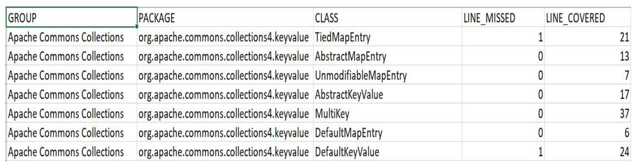
Statement coverage is a white box testing technique, which involves the execution of all the statements at least once in the source code. It is a metric, which is used to calculate and measure the number of statements in the source code which have been executed. Using this technique we can check what the source code is expected to do and what it should not. It can also be used to check the quality of the code and the flow of different paths in the program. The main drawback of this technique is that we cannot test the false condition in it.

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

It is applied at the class level.

Data Collection

Jacoco tool is used for the calculation. Plugin for Jacoco is added to the pom file in the maven project and the project is run as a maven test to get the jacoco.csv and index.html file containing class level results for the number of lines missed and lines covered. Result for the percentage is calculated considering the covered lines.



GROUP	PACKAGE	CLASS	LINE_MISSED	LINE_COVERED
Apache Commons Collections	org.apache.commons.collections4.keyvalue	TiedMapEntry	1	21
Apache Commons Collections	org.apache.commons.collections4.keyvalue	AbstractMapEntry	0	13
Apache Commons Collections	org.apache.commons.collections4.keyvalue	UnmodifiableMapEntry	0	7
Apache Commons Collections	org.apache.commons.collections4.keyvalue	AbstractKeyValue	0	17
Apache Commons Collections	org.apache.commons.collections4.keyvalue	MultiKey	0	37
Apache Commons Collections	org.apache.commons.collections4.keyvalue	DefaultMapEntry	0	6
Apache Commons Collections	org.apache.commons.collections4.keyvalue	DefaultKeyValue	1	24

[Figure 1] Snapshot of Jacoco Test Coverage Report for Statement Coverage

Advantages

- It verifies what the written code is expected to do and not to do
- It measures the quality of code written
- It checks the flow of different paths in the program and it also ensures that whether those paths are tested or not.

Disadvantages

- It cannot test the false conditions.
- It does not report whether the loop reaches its termination condition.

B- 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.

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 behavior of the application.

Data Collection

Jacoco tool is used for the calculation. Plugin for Jacoco is added to the pom file in the maven project and the project is run as a maven test to get the jacoco.csv and index.html file containing class level results for the number of branches missed and branches covered. Result for the percentage is calculated considering the covered branches.

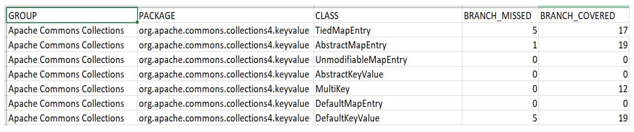
It is applied at the class level.

Advantages

- To validate that all the branches in the code are reached.
- To ensure that no branches lead to any abnormality of the program’s operation.

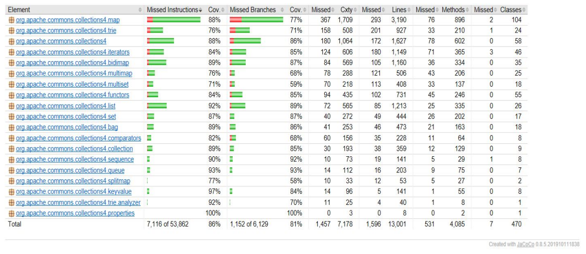
Disadvantages

- There may be other conditions that can be used for decision making.
- This metric ignores branches within boolean expressions which occur due to short-circuit operators.



GROUP	PACKAGE	CLASS	BRANCH_MISSED	BRANCH_COVERED
Apache Commons Collections	org.apache.commons.collections4.keyvalue	TiedMapEntry	5	17
Apache Commons Collections	org.apache.commons.collections4.keyvalue	AbstractMapEntry	1	19
Apache Commons Collections	org.apache.commons.collections4.keyvalue	UnmodifiableMapEntry	0	0
Apache Commons Collections	org.apache.commons.collections4.keyvalue	AbstractKeyValue	0	0
Apache Commons Collections	org.apache.commons.collections4.keyvalue	MultiKey	0	12
Apache Commons Collections	org.apache.commons.collections4.keyvalue	DefaultMapEntry	0	0
Apache Commons Collections	org.apache.commons.collections4.keyvalue	DefaultKeyValue	5	19

[Figure 2] Snapshot of Jacoco Test Coverage Report for Branch Coverage



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.collections4.map		88%		77%	367	1,709	293	3,190	76	896	2	104
org.apache.commons.collections4.trie		76%		71%	158	508	201	927	33	210	1	24
org.apache.commons.collections4		88%		86%	180	1,064	172	1,627	78	602	0	58
org.apache.commons.collections4.iterators		84%		85%	124	606	180	1,149	71	365	3	46
org.apache.commons.collections4.bidimap		89%		87%	84	569	105	1,160	36	334	0	35
org.apache.commons.collections4.multimap		76%		68%	78	288	121	506	43	206	0	25
org.apache.commons.collections4.multiset		71%		59%	70	218	113	408	33	137	0	18
org.apache.commons.collections4.functors		84%		85%	94	435	102	731	45	246	0	55
org.apache.commons.collections4.list		92%		89%	72	565	85	1,213	25	335	0	26
org.apache.commons.collections4.set		87%		87%	40	272	49	444	26	202	0	17
org.apache.commons.collections4.bag		89%		86%	41	253	46	473	21	163	0	18
org.apache.commons.collections4.comparators		82%		68%	60	156	35	228	11	64	0	8
org.apache.commons.collections4.collection		89%		85%	30	193	38	359	12	129	0	9
org.apache.commons.collections4.sequence		90%		92%	10	73	19	141	5	29	1	8
org.apache.commons.collections4.queue		93%		93%	14	112	16	203	9	75	0	7
org.apache.commons.collections4.splitmap		77%		58%	10	33	12	53	5	27	0	2
org.apache.commons.collections4.keyvalue		97%		84%	14	96	5	141	1	55	0	8
org.apache.commons.collections4.trie.analyzer		92%		70%	11	25	4	40	1	8	0	1
org.apache.commons.collections4.properties		100%		100%	0	3	0	8	0	2	0	1
Total	7,116 of 53,862	86%	1,152 of 6,129	81%	1,457	7,178	1,596	13,001	531	4,085	7	470

Created with JaCoCo 0.8.5.201910111838

[Figure 3] Snapshot of Jacoco Test Coverage Report for Branch Coverage

C- Mutation Coverage Metric:

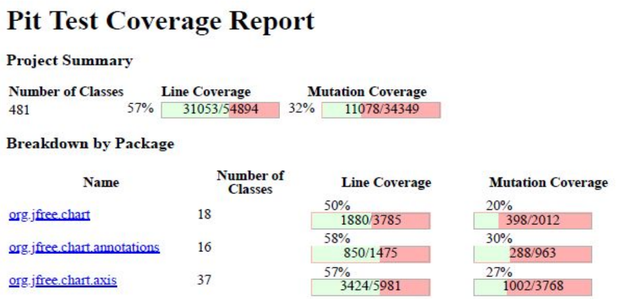
Software testing is an important part of the software engineering lifecycle process. It plays a pivotal role in the quality assurance of the software product. During software maintenance, testing is a crucial activity to ensure the quality of program code as it evolves over time. With the increasing 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 case is unable to detect fault it will be hard to build a good software. This is fulfilled by Mutation Testing.

It's a Whitebox-testing used to develop the hidden test cases by modifying code and checking 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 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 the Pit-Test tool. This tool can be installed in Eclipse by the name of Pitclipse. To execute this tool, we need to run the project under PIT configuration of the PIT test configuration. The results are finally generated in an Html file which is called the Pit report.



Pit Test Coverage Report

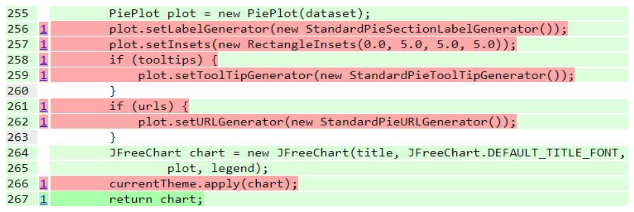
Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
481	57%	31053/54894	32%	11078/34349

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
org.jfree.chart	18	50% 1880/3785	20% 398/2012
org.jfree.chart.annotations	16	58% 850/1475	30% 288/963
org.jfree.chart.axis	37	57% 3424/5981	27% 1002/3768

Mutations performed during Test Coverage:



```
            PiePlot plot = new PiePlot(dataset);
1           plot.setLabelGenerator(new StandardPieSectionLabelGenerator());
1           plot.setInsets(new RectangleInsets(0.0, 5.0, 5.0, 5.0));
1           if (tooltips) {
1               plot.setToolTipGenerator(new StandardPieToolTipGenerator());
            }
1           if (urls) {
1               plot.setURLGenerator(new StandardPieURLGenerator());
            }
            JFreeChart chart = new JFreeChart(title, JFreeChart.DEFAULT_TITLE_FONT,
                    plot, legend);
1           currentTheme.apply(chart);
1           return chart;
```

D- Cyclomatic Complexity Metric:

In the software development process, there are many difficulties to analyze the quality of code. To measure one kind of quality for this task is Cyclomatic complexity. There are fundamentally two ways to identify the complexity. One among them is the essential complexity which is inevitable. The second one is the accidental complexity which is the byproduct of development done in hurry. This can be measured using the McCabe complexity metric which is also known as Cyclomatic complexity. This is one of the simplest metrics to calculate accidental complexity.

The mathematical method to make the program measured is also done by Cyclomatic Complexity.Proposed a formula to calculate the complexity:

$$\mathbf{CC = E - N + 2P} \quad \text{or} \quad \mathbf{CC = D + 1}$$

Where,

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

Pros of Cyclomatic Complexity:

● It calculates the total possible linearly independent paths executed for a component.

● The number of independent paths can be used to determine the number of test cases needed to cover the component.

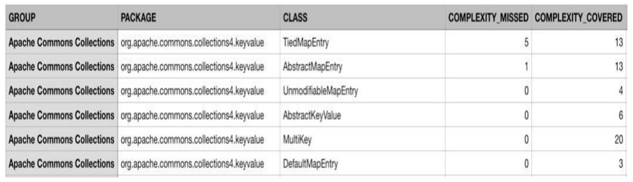
Cons of Cyclomatic Complexity:

● It considers all the control predicates as the same level of complexity. E.g. for loop and if statements can have different levels of complexity.

● It doesn't take the nesting into account when calculating from the independent paths.
There are several approaches to measure the project and one of the simplest yet a powerful tool to measure. Among the structural or the quantitative approach, quantitative is simple counting of code lines and the project size. This simple approach applied under the different schemes can be very useful to analyze the complexity of the project.

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.

In this project, the Cyclomatic complexity metric is calculated using the Jacoco plugin. As described in the JaCoCo documentation, JaCoCo calculates complexity for each non-abstract method and summarizes complexity for classes, packages and groups. Below is the screenshot generated by the JaCoCo plugin where the covered complexity and missed complexity column represents the Cyclomatic complexity given class wise.



GROUP	PACKAGE	CLASS	COMPLEXITY_MISSED	COMPLEXITY_COVERED
Apache Commons Collections	org.apache.commons.collections4.keyvalue	TiedMapEntry	5	13
Apache Commons Collections	org.apache.commons.collections4.keyvalue	AbstractMapEntry	1	13
Apache Commons Collections	org.apache.commons.collections4.keyvalue	UnmodifiableMapEntry	0	4
Apache Commons Collections	org.apache.commons.collections4.keyvalue	AbstractKeyValue	0	6
Apache Commons Collections	org.apache.commons.collections4.keyvalue	MultiKey	0	20
Apache Commons Collections	org.apache.commons.collections4.keyvalue	DefaultMapEntry	0	3

***E-Adaptive Maintenance Effort Model*:**

A software continues goes under the enhancement which requires team effort to adapt the new functionality and keep the original product working. Unfortunately, developers and managers underestimate the planning and estimation required to make the required changes. To overcome this problem, an adaptive maintenance effort model was developed that uses software metrics to estimate the cost of software releases.

To calculate the effort, we have used changes in the lines of code (DLOC) as a metric in the regression model. The DLOC can be determined from the difference in the versions of the software. Once the DLOC is determined, we can calculate maintenance effort by using this model:

E = -40 + 6.56DLOC

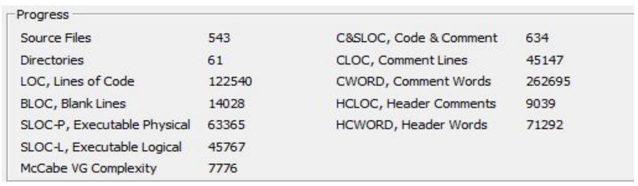
Where,

E is the maintenance effort in person-hours

DLOC is the difference in lines of code.

This effort can also be used for regression analysis. Such analysis can be helpful to predict the future effort in machine learning model. It can also be useful to comprehend the relationship between the variable used and the effort during the maintenance process.

To calculate the DLOC we have used LOCMetric tool. This tool when run of the product produces multiple metrics mentioned in the figure (1) below:

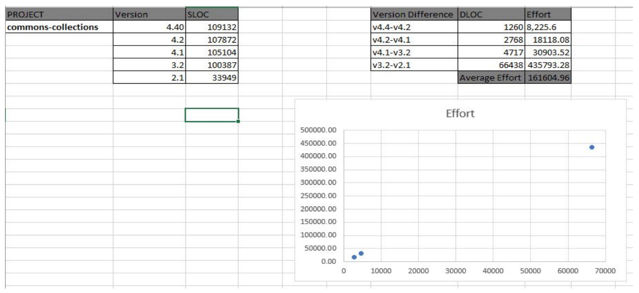


Progress

Source Files	543	C&SLOC, Code & Comment	634
Directories	61	CLOC, Comment Lines	45147
LOC, Lines of Code	122540	CWORD, Comment Words	262695
BLOC, Blank Lines	14028	HCLOC, Header Comments	9039
SLOC-P, Executable Physical	63365	HCWORD, Header Words	71292
SLOC-L, Executable Logical	45767		
McCabe VG Complexity	7776		

From the metrics shown in the fig (1), we have used SLOC-P and SLOC-L and calculated the sum (SLOC) of them to get the source lines of code of the project. Similarly, this operation is performed on all the versions of the product. Further, the DLOC is calculated by subtracting the SLOC of two consecutive versions of the product. At last, the effort is calculated by applying the -40 +6.56DLOC formula.

The result of DLOC and effort is captured in excel sheet as shown in figure (2) below:



PROJECT	Version	SLOC
commons-collections	4.40	109132
	4.2	107872
	4.1	105104
	3.2	100387
	2.1	33949

Version Difference	DLOC	Effort
v4.4-v4.2	1260	8,225.6
v4.2-v4.1	2768	18118.08
v4.1-v3.2	4717	30903.52
v3.2-v2.1	66438	435793.28
	Average Effort	161604.96

F-Software Quality Metric (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 to as post-release defects. It is defined as the number of defects confirmed in software/module during a specific period of operation or development divided by the size of the software/module. It is counted per thousand lines of code also known as KLOC. Defect density is used as the indicator for the Product Quality.

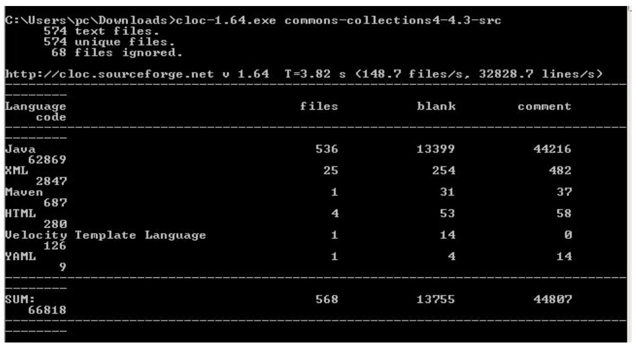
Defect Density = Defect Count / Size of the Software

The defect density can be calculated by the type of issues found in the issue tracking system of the Projects. We only take in consideration the issues found out by the in-house team of the project which they track using their respective issue tracking system. This metric also depends on the quality of the testing team as well. There are multiple kinds of issue like Wish or New Feature, which are not bugs, but the features or pre-release issues that developers usually work on. 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 is testing for the runtime defects in the system. If any anomaly is found, it is tracked by identifying it as a bug.

Data Collection

We need the source lines of code to calculate the size of the system. There are many tools available to gather this data. But we have used cloc (https://github.com/AlDanial/cloc/). To run cloc on Windows computers, one must first open up a command (aka DOS) window and invoke **cloc**.exe from the command line there.

The data collection using this tool is illustrated as below:

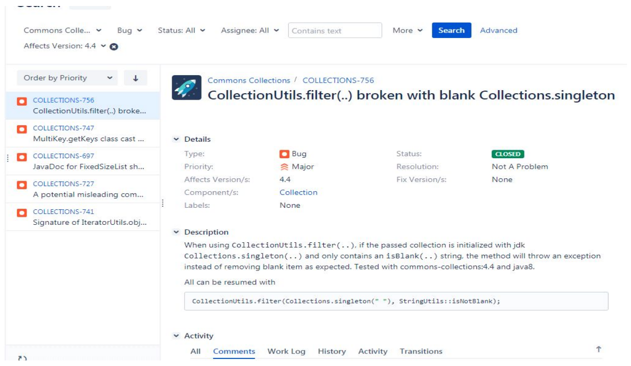


```
C:\Users\pc\Downloads>cloc-1.64.exe commons-collections4-4.3-src
     574 text files.
     574 unique files.
      68 files ignored.

http://cloc.sourceforge.net v 1.64  T=3.82 s (148.7 files/s, 32828.7 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment
   code
-------------------------------------------------------------------------------
Java                           536          13399          44216
   62869
XML                             25            254            482
   2847
Maven                            1             31             37
   687
HTML                             4             53             58
   280
Velocity Template Language       1             14              0
   126
YAML                             1              4             14
     9
-------------------------------------------------------------------------------
SUM:                           568          13755          44807
   66818
-------------------------------------------------------------------------------
```

[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 using their respective Jira Reports from the user interface and then export them as a CSV from the easier calculation. In order to achieve this, we first have to filter the issues on JIRA Tracking System by the type "Bug" (the one with the red icon). And we can apply another filter with the value 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. Below Figure Illustrates Jira Report after filtering out Bugs for a particular Version of a Project.



COLLECTIONS-756
CollectionUtils.filter(..) broke...
COLLECTIONS-747
MultiKey.getKeys class cast ...
COLLECTIONS-697
JavaDoc for FixedSizeList sh...
COLLECTIONS-727
A potential misleading com...
COLLECTIONS-741
Signature of IteratorUtils.obj...

Commons Collections / COLLECTIONS-756
CollectionUtils.filter(..) broken with blank Collections.singleton

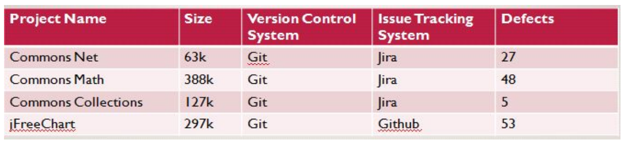
Details
Type: Bug
Priority: Major
Affects Version/s: 4.4
Component/s: Collection
Labels: None
Status: CLOSED
Resolution: Not A Problem
Fix Version/s: None

Description
When using `CollectionUtils.filter(..)`, if the passed collection is initialized with jdk `Collections.singleton(..)` and only contains an `isBlank(..)` string, the method will throw an exception instead of removing blank item as expected. Tested with commons-collections:4.4 and java8.

All can be resumed with

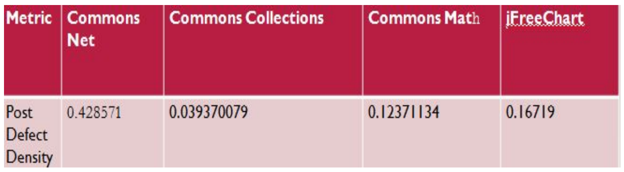
```
CollectionUtils.filter(Collections.singleton(" "), StringUtils::isNotBlank);
```

Activity
All
Comments
Work Log
History
Activity
Transitions



Project Name	Size	Version Control System	Issue Tracking System	Defects
Commons Net	63k	Git	Jira	27
Commons Math	388k	Git	Jira	48
Commons Collections	127k	Git	Jira	5
jFreeChart	297k	Git	Github	53

[Table 1] Jira Report



Metric	Commons Net	Commons Collections	Commons Math	jFreeChart
Post Defect Density	0.428571	0.039370079	0.12371134	0.16719

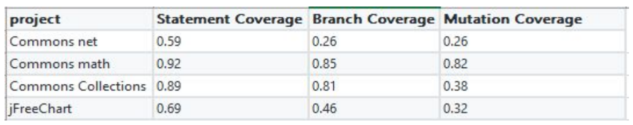
[Table 2] Post Release Defect Density for all the Test Subjects

As we can see from the table above, the number of bugs is not always proportional to the size of code. It can be proportional 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.

Correlations

- ***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.



project	Statement Coverage	Branch Coverage	Mutation Coverage
Commons net	0.59	0.26	0.26
Commons math	0.92	0.85	0.82
Commons Collections	0.89	0.81	0.38
jFreeChart	0.69	0.46	0.32

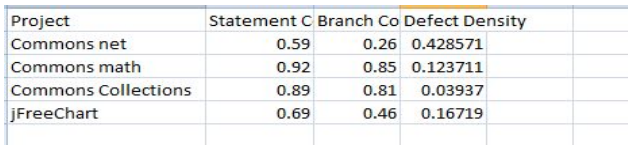
[Table] Correlation between mutation coverage and Coverage Metrics

From the table, we can see that Code coverage is moderately correlated to Mutation score for the project. Other than that, the analysis shows 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 test suits. The analysis for the branch coverage is also like the statement coverage. It is also included in the replication package along with this paper. Our results conclude that the Spearman correlation between statement coverage and mutation coverage is 1.0, Pearson correlation is 0.75, and Pearson correlation between branch coverage and mutation coverage is 1.0, Pearson correlation is 0.74. It indicates increasing monotonic trends between statement, branch coverage and mutation coverage and justifies our rationale given above.

- ***B Correlation between each coverage metric and software quality metric.***

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

As per the below table, we considered the latest version of the system in issue tracking system as well as the code.



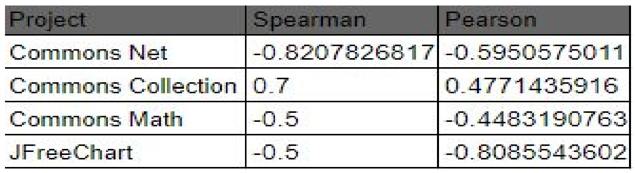
Project	Statement C	Branch Co	Defect Density
Commons net	0.59	0.26	0.428571
Commons math	0.92	0.85	0.123711
Commons Collections	0.89	0.81	0.03937
jFreeChart	0.69	0.46	0.16719

[Table] 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 depends 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-0.799, Pearson correlation is -0.8644, and Pearson correlation between branch coverage and the post release defect density is -0.799, Pearson correlation is -0.8824.

- ***C Correlation between software quality metric and software maintenance effort***

The relation behind this correlation is that the more effort we make to maintain the programme, the less deficiencies we can experience after release.



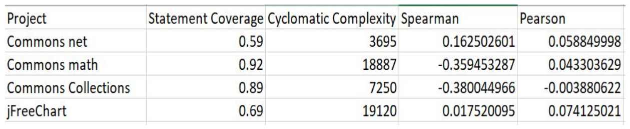
Project	Spearman	Pearson
Commons Net	-0.8207826817	-0.5950575011
Commons Collection	0.7	0.4771435916
Commons Math	-0.5	-0.4483190763
JFreeChart	-0.5	-0.8085543602

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

From the table, we can see the correlation coefficient between both the effort and the Defect Density of each project.Inferencing from the table we can strongly suggest that maintaining 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 and the Pearson correlation between both the metrics during our analysis which shows the importance of the maintenance efforts in the software development.

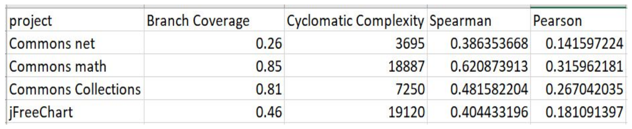
- ***D Correlation between each coverage metric and complexity metric.***

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.



Project	Statement Coverage	Cyclomatic Complexity	Spearman	Pearson
Commons net	0.59	3695	0.162502601	0.058849998
Commons math	0.92	18887	-0.359453287	0.043303629
Commons Collections	0.89	7250	-0.380044966	-0.003880622
jFreeChart	0.69	19120	0.017520095	0.074125021

[Table 7] Results for statement coverage and cyclomatic complexity



project	Branch Coverage	Cyclomatic Complexity	Spearman	Pearson
Commons net	0.26	3695	0.386353668	0.141597224
Commons math	0.85	18887	0.620873913	0.315962181
Commons Collections	0.81	7250	0.481582204	0.267042035
jFreeChart	0.46	19120	0.404433196	0.181091397

[Table 8] Results for branch coverage and cyclomatic complexity

From table 7 and 8, we see that the Spearman, Pearson coefficient have quite less values and in some projects they are negative which indicates decreasing monotonic trends between statement, branch coverage and cyclomatic complexity and justifies our rationale given above.

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 the development team.

- In large projects, 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 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.

IV. TOOLS AND CALCULATIONS

JaCoCo

The JaCoCo is developed by the 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. The configuration steps are mentioned along with the replication package.

CLOC

We have used cloc tool to calculate the Source Lines of Code. It is developed by AlDanial and hosted on github. 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.The detailed guideline to use this tool is available here (https://github.com/AlDanial/cloc/)

PIT-Testing tool

We have used the 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. 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..

Correlations (Python)

We have Developed a Python program to perform the calculation on the data that we have gathered using the above tools. Python scipy library has built-in functions to calculate the correlation between two data sets.

V. 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.

- 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 suggest 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.

VI. 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.

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

2.Coverage.Obtained from: "http://www.linozemtseva.com/research/2014/icse/coverage/coverage_paper.pdf"

3.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"

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.pdWhat Are Software Metrics and How Can You Track Them?