An Empirical Study on Relation Between Object-Oriented Metrics and Change-Proneness

Tanisha Shah
Dept. of Engineering & Computer Science
Concordia University
Montreal, Canada
shah.tanisha22@gmail.com

Deep Patel
Dept. of Engineering & Computer Science
Concordia University
Montreal, Canada
deeppatel2223@gmail.com

Jemish Paghadar
Dept. of Engineering & Computer Science
Concordia University
Montreal, Canada
jemish271297@outlook.com

Raj Patel
Dept. of Engineering & Computer Science
Concordia University
Montreal, Canada
rajp1908@gmail.com

Hardik Vora
Dept. of Engineering & Computer Science
Concordia University
Montreal, Canada
hardik7304@gmail.com

Dolly Modha
Dept. of Engineering & Computer Science
Concordia University
Montreal, Canada
modhadolly@gmail.com

***Abstract*— In software development, evolution of software is the most important part of it. Evolution of software means developing new versions with changes that leads to corrective and adaptive maintenance. Change proneness is an external quality attribute. It depicts the changes in classes across different version of the system. If changes are predicted earlier than errors and bugs can be prevented. In this paper, we describe the comparison of different software systems such as Eclipse, Intellij and NetBeans and co-relate the change proneness in the software during their evolution using the C&K metrics, QMOOD metrics. Additionally, we also analyzed the attributes like understandability, functionality, reusability, extendibility that are affected due to change proneness. For concluding how change-prone, a class is we computed maintainability index and using linear regression, we calculated the prediction power.**

***Keywords*— Change-proneness, software metrics, software quality, regression, maintainability index.**

I. INTRODUCTION

In software industry, the major factors are the time and cost for developing and maintaining a software. In addition, of all merits of the object-oriented paradigm, evolution is probably the most important in a world of constantly changing requirements [12]. Evolution directly relates to the economic part of a software development as most of the time and money is invested in maintaining the software. Practically, the addition of new functionality i.e. evolution in an object-oriented system should have as limited impact on existing code as possible. If the modification of a class method imposes code changes to several existing classes, change-proneness is of limited value [12]. In other words, change-proneness is defined as how flexible a class is for future changes without producing any error and affecting the quality of software. Predicted change-prone class increases the maintenance and testing of the software.

Motivation: A lot of research is done on predicting the change-proneness using metrics, but the results concluded

are not same. For one, if one metric shares a strong correlation with change-proneness while for other study it does not. Thus, there is still scope for more conclusions using different metrics.

Premise: Although the existence of many works on change-proneness and software quality metrics, no previous work has extended the empirical investigation of the change-proneness of classes with object-oriented metrics and software quality attributes to a larger set of system, to study the impact of change-proneness on this aspect of software evolution.

Goal: Our goal is to find which metrics have a relation with change-proneness. We will investigate whether all the C&K metrics are related or not and which of the QMOOD metrics are related to change-proneness. We will also discuss which of the six quality attributes of software affects with a class being change-prone or not.

Contribution: Our study is performed using C&K metrics and QMOOD metrics for three releases of Eclipse, IntelliJ, and NetBeans. We used 10 metrics in total. Basis of our results is at both class level and system level. We show that if a model fits properly then some metrics have a strong correlation with change-proneness.

Relevance: Comprehension of change proneness in the software during their evolution using the C&K metrics, QMOOD metrics and analyzing the affected software quality attributes is important from the points of view of both researchers and practitioners. Therefore, this study justifies a posteriori previous work on change proneness: within the limits of the threats to its validity, there is significant relationship between change-proneness and object-oriented metrics and there is an effect on quality measures as well and therefore it may indeed hinder software evolution.

Organization: Later, in this paper, we will cover the related work study and comparison with our project in SECTION II, then SECTION III summarizes about the independent variables (C&K metrics and QMOOD metrics) and dependent variables along with their implementation details. In addition, the hypotheses are defined. In SECTION IV, we will talk about the statistical analysis conducted to accept or reject the hypotheses, then SECTION V, VI describes about the results obtained. In SECTION VII, we will discuss about the threats to validity regarding our project. Lastly, in SECTION VIII, we discussed the conclusions.

II. RELATED WORK

There are various studies performed on the investigation of the change-proneness in object-oriented software. In this section, we will therefore present a summary of five contributions that are related to our work.

"An Empirical Study of the Relationships between Design Pattern Roles and Class Change Proneness" [1]

This paper presents a study of the evolution of classes playing different roles in design motifs to understand whether there are roles that are more change-prone than others and there are some changes that occur more to certain roles than to others. Kinds of changes include: change to method implementation, method or attribute addition/removal, and extension by sub classing. Their main motivation was that only considering occurrences of the motifs as a "whole" and that design motif identification was performed using the DeMIMA (Design Motif Identification Multi-layered Approach) which not only identifies the classes, but also the different roles played by classes in occurrences of motifs. This paper also reports new evidence on kind and frequency of changes which confirms the intuitive behavior – e.g., concrete classes tend to be more subject to changes than abstract classes. Their results suggest that developers must carefully design classes playing within a motif role that will likely be subject to frequent changes and kinds of changes that will likely impact elsewhere in the system.

This study is similar to ours in terms of examining the change proneness between systems, but we are not considering any design motifs. There are, however, many cases where the actual, observed changes deviates from the intuition and form the common wisdom. For example, in the Composite motif classes playing the role of Composite can be more complex than expected and because of that undergoing a high number of changes. In our study, we try to extend the empirical investigation to a larger set of systems, to investigate on different roles co-changes, and to find evidence of correlations between kind of changes.

"Understanding change-proneness in OO software through visualization" [4]

Main goal of this research is to identify and visualize classes and class interactions that are the most change-prone. The problem that is being solved is in knowing where those change-prone clusters are can help focus attention, identify targets for re-engineering and thus provide product-based information to steer maintenance processes. This work tracks design changes based on common data interactions and the calling structure in COBOL programs. The aim is to see how the design structure of a system can affect the change-proneness of individual classes. They also examined the case study data focusing on the classes that are the most change-

prone, distinguishing between local changes and change coupling between classes. The approach of this study is to analyze both the implementation structure of the software system and change logs by finding patterns and change-proneness measures. The analysis of the implementation structure provides a characterization of the individual classes in the system and identifies the design patterns. The change logs provide the information needed to develop a change architecture for the system; they identify individual changes and all classes that were changed for each reported change. We learned how to quantify the degree to which classes are change-prone both locally and in their interactions with others from this study.

The similarities of both the study is that both emphasizes on the change proneness of the system and uses QMOOD metrics and C&K metrics. However, we are emphasizing on the system built in java and have broaden the metrics while this paper was emphasized on system built in C++ and have taken limited metrics. In our paper, we try to improve the results by considering almost all metrics related to change-proneness leading to better results.

"Thresholds for Size and Complexity Metrics: A Case Study from the Perspective of Defect Density" [2]
The primary goal of this research is to be able to provide guidelines to practitioners on what kind of code has better quality. They aim to observe if a consistent relationship between metric thresholds and software quality is present in Open Source Software and industrial projects. They replicated published technique for calculating metric thresholds to determine high-risk files based on code size and complexity using a very large set of open and closed source projects written primarily in Java and evaluated software quality using two criteria - defect proneness (probability of a file having a defect), and defect density (the number of defects/LOC) on three OSS and four industrial projects. These findings confirm results that the size bench marks are associated with high defect-proneness. It also indicates that the size is a better threshold of defect-proneness than complexity.

In our work, we propose to compare different software systems like Eclipse, Intellij and NetBeans and corelate the change proneness in the software during their evolution using the software metrics. They have also used total LOC in a file and cyclomatic complexity to evaluate defect-proneness. However, their findings have immediate practical implications as well like the redistribution of Java code into smaller and less complex files may be counterproductive. They have only considered metrics Lines of code, Module Interface size and cyclomatic complexity for software quality evaluation whereas we have worked on more metrics and considered quality measures as well.

"Investigating the OO Characteristics of Software using CKJM Metrics" [6]
The objective of this paper is to validate software product metrics in software engineering using existing tools with the help of statistical analysis. This paper calculates and analyses 12 object-oriented software metrics to improve the software development process and quality. These metrics were tested for 10 sample open-source programs written in java. The frequency and descriptive analysis were then performed on the obtained results using SPSS tool. This empirical validation provides the practitioner with some empirical evidence indicating that most of the metrics can be helpful indicators of quality. Furthermore, most of the metrics were found to be redundant indicators even on being comparatively independent from each other. The results thus obtained endow with motivation for additional investigation and refinement of C&K OO metrics. Hence, we have used these metrics to investigate object-oriented characteristics in our project.

"QMOOD metric sets to assess quality of java program" [5]
This paper describes the model to evaluate and grade the java programs, based on QMOOD (Quality Model for Object Oriented Design) which is the hierarchical model that defines relation between qualities attributes (like reusability, functionality, effectiveness, understand ability, extendibility, flexibility) and design properties with the help of equations. They computed all the design metrics from the java program taken as input and normalized all the values of the design metrics and from that they found all the design properties which will help the developers to directly decide the best design. This paper also concludes that it's not necessary that every time increasing the positive parameter would result in better quality. Therefore, we have used QMOOD metrics to measure the quality as they are good indicators of quality of software and they are computed early in the design phase that helps in reducing complexity at the later stages.

III. STUDY DEFINITION

The *goal* is to study about the relationship between the object-oriented metrics and change-proneness.
The *quality focus* is decrease in effort and cost for solving the bugs produced after the changes and maintaining the quality of the software.
Our study is useful for both the software developers and the researchers. From the *perspective* of developers-already known change-prone classes is useful in maintaining the software [13]. It makes evolution less

fault-prone. While, for researchers it is helpful in increasing their understanding for the software and for the impact.

The *context* for this study consists of different releases of three systems, Eclipse, IntelliJ and NetBeans. Eclipse[3] is an open-source integrated development environment that is mainly written in Java and C++. IntelliJ IDEA[2] is a Java integrated development environment for developing computer software. Developed by JetBrains and is available as an Apache 2 Licensed community edition, and in a proprietary commercial edition. Both is used for commercial development. NetBeans[1] is an integrated development environment for Java. NetBeans allows applications to be developed from a set of modular software components called modules.

Our main reason for selecting this software is that all three projects are written in the same language. In addition, all this software has more than 200 files or more. These systems completely relate to our goal of co-relating change proneness with different versions using different metrics as this software have a vast range for selecting versions. Additionally, the releases selected have gap of one year. The details for each project are as follows:

ECLIPSE
https://github.com/eclipse/eclipse.platform.ui-*source code*
Releases:
eclipse.platform.ui-master released on 8 March,2019
eclipse.platform.ui-I20180601-0915 released on 1 June, 2018
eclipse.platform.ui-I20170501-2000 released on 1 May, 2017

NETBEANS
https://github.com/apache/incubator-netbeans-*source code*
Releases: incubator-netbeans-master released on 9 March, 2019
incubator-netbeans-9.0-vc3 released on Jul 8, 2018
netbeans-releases-jshell_api9 released on Nov 17, 2017

INTELLIJ
https://github.com/JetBrains/intellij-community-*source code*
Releases: IntelliJ IDEA Community Edition released on 9 March, 2019
webstorm/181.5540.36 released on Nov 19, 2018
webstorm/172.2465.2 released on May 23, 2017

[1] https://en.wikipedia.org/wiki/NetBeans
[2] https://en.wikipedia.org/wiki/IntelliJ_IDEA
[3] https://en.wikipedia.org/wiki/Eclipse(software)

A. Research Questions

RQ: What will be the impact of software quality metrics in predicting change-proneness in software development and evolution?

Null Hypotheses: There is no relation between the C&K metrics, QMOOD metrics and maintainability index (Change-proneness).
Alternative Hypotheses (One-sided): There is some relation between the C&K metrics, QMOOD metrics and maintainability index.

We evaluate the C&K metrics and QMOOD metrics to define our suite hypotheses [14].

- DIT - Depth of Inheritance Tree
  - We believe that more inheritance meaning more dependency between classes. For that reason, a class will need more maintenance for changes.
- WMC - Weighted Methods per Class
  - Since WMC relates to complexity, which relates to size thus we assume that it relates to changes. A class with high WMC is very likely to have be lesser change-prone.
- NOC - Number of Children
  - NOC is similarly to DIT. A class with many children may need too many changes thus making it less change-prone.
- CBO - Coupling Between Objects
  - CBO is a metric with high values when more coupling is there between classes. This relates to the code structure and high values can be from bigger classes or classes that are more complex. That means it could potentially need more changes. Thus, it is less change-prone.
- RFC - Response for a Class
  - RFC relates to the methods of a class. A class that has more methods can have a higher RFC. That relates to the size of a class, which we expect to be related to the number of changes concluding it to be less change-prone.
- LCOM - Lack of Cohesion of Methods
  - A class with high lack of cohesion can have more code than is necessary. Classes would need better structure to be with higher cohesion. This means that the changes are higher making it less change-prone.
- NOM – Number of methods
  - A class with higher number of total methods of its own will have higher NOM value. This means that while making changes in a class more effort it required.
- LCAM – Lack of Cohesion among methods

- LCAM have higher value when methods do not have cohesion among themselves resulting in a more number of changes in future.
- LOC – Line of Code
  - When line of code is higher means bigger the code. Hence, resulting in requiring more changes for the code.

B. Variable Selection (Examined Variables)

In this study, we are relating the dependent variable with the independent variable to accept or reject the hypotheses. The independent variables are all the C&K metrics and few QMOOD metrics. On the other end, maintainability index is our dependent variable.

Independent Variables:

The independent variables represent the internal quality of software system. They are also known as predictor variable. It is useful in understanding the relation with our dependent variable. In our study, we have nine independent variables. The C&K metrics are: Weighted Method per Class (WMC), Coupling between Objects (CBO), Lack of Cohesion of Methods (LCOM), Response Set of a Class (RFC), Depth Inheritance Tree (DIT), Number of Children (NOC). While, the QMOOD metrics are: Number of Methods (NOM), Lack of Cohesion among Methods (LCAM), Line of Code (LOC).

Dependent Variable:

The dependent variable represents the external quality of software system. It is also known as response variable as it is measured or observed from the independent variable. In our study, the dependent variable is maintainability index.

Tools used:

In this section, we discuss about the projects and tools, which we used to collect the metric values. The tool that we have used for computing independent variable is CodeMR and Understand while for dependent variable we have used JHawk.

Understand:

- Understand™ is a powerful source code comprehension tool.
- It is used to quickly understand large or complex legacy code bases ... often with poor documentation.
- Programmers and team leaders regularly use it to visualize complex legacy code, perform impact analysis, and deliver powerful metrics.
- Perform impact analysis for changes and evaluates metrics values.
- Understand creates a repository of the structures and relations in a software project which it can help us to a better comprehension of the source code.

CodeMR:

- CodeMR is a software quality and static code analysis tool that helps to developing better quality code and supports multiple programming languages.
- CodeMR is integrated with Eclipse and IntelliJ IDEA which Supports Java, Kotlin, Scala and C++ languages. CodeMR static code analysis tool extracts and visualizes code in a big picture at Metric Distribution, Package Structure, TreeMap, Sunburst and Dependency views and Project Outline.
- It visualizes high-level Object-Oriented quality attributes and low-level metrics.
- It extracts quality metrics and all types of relations directly from source code. You can see which parts need refactoring.
- It also Check code complexity, cohesion and coupling while investigating code quality.
- CodeMR has advanced modularization algorithms.
- It shows current package structure or extracts modules and services from source code.
- CodeMR helps to transform microservice architecture from their monolithic applications.

JHawk:

JHawk is a static code analysis tool - i.e. it takes the source code of your project and calculates metrics based on numerous aspects of the code - for example volume, complexity, relationships between class and packages and relationships within classes and packages. Using JHawk tool we have calculated maintainability index.

Metrics Definition:

1) Weighted Method per Class (WMC): -

WMC [6] measures the overall complexity of a class. It is the sum of cyclomatic complexity of each method for all methods of a class.

- $WMC = \sum_{i=1}^{n} CC_i$

where,

n is the number of methods in a class

CCi is the cyclomatic complexity of method i

A high value of WMC suggest that it is more prone to errors and less like to be reusable. WMC might be helpful to understand whether the class is change prone or not for future developments.

IMPLEMENTATION:

The logic behind calculating WMC is first counting the number of control statements in each method and then adding the value of all the methods. We have calculated

WMC using the CodeMR tool, which is used for static code analysis. We have also calculated the data using Understand tool. WMC is calculated at package and class level for four releases of all the three software systems (Eclipse, NetBeans, and IntelliJ).

2) Depth of Inheritance Tree (DIT): -

DIT [6] is useful in calculating the inheritance of a class. DIT for a class X is the maximum path length from X to root of inheritance tree. The deeper a class is in the hierarchy, the more methods and variables it is likely to inherit, making it more complex. Deep trees as such indicate greater design complexity. The higher the value of DIT higher chances of change proneness due to inheritance related design violations. Nominal range of DIT is 0-4. If value of DIT is zero, which indicates that, there is no inheritance for that class. If the value is higher than 4 then it means that it will increase the encapsulation, which will eventually lead to problems when changes made to classes.

IMPLEMENTATION:

The logic behind calculating the DIT is by counting the number of super classes that extends a class. DIT has been calculated in understand and codeMR both the tools. In Understand, DIT is calculated as MaxInheritanceTree. From understand we have collected DIT as class level and package level.

3) Number of immediate subclasses (NOC): -

NOC [6] is the number of subclasses subordinated to a class in the hierarchy. The NOC for class X is the number of immediate subclasses of X in the inheritance tree. The higher the value of NOC, the higher reuse of the base class. However, if base class contains error, these will propagate to many subclasses. From NOC we will able to analyze the reusability of the code of class.

IMPLEMENTATION:

The logic behind calculating NOC is counting the number of classes inheriting the given class. NOC is implemented by at class level and package level in understand. In understand NOC is implemented as CountClassDerived. In CodeMR, NOC is calculated at class level.

4) Coupling between objects (CBO): -

Coupling between objects (CBO) [6] is a count of the number of classes that are coupled to a class i.e. where the methods of one class call the methods or access the variables of the other. These calls need to be counted in both directions, so the CBO of class A is the size of the set of classes that class A references and those classes that reference class A. Since this is a set - each class is counted only once even if the reference operates in both directions i.e. if A references B and B references A, B is only counted once. Ideally the value of CBO must as low as possible.

IMPLEMENTATION:

The logic behind calculating the CBO is implemented by computing the various calls made from class to other classes then counting the other classes using that class. In understand CBO is calculated as CountClassCoupled. The higher value of CBO indicated that class is more change prone thus increasing difficulty for maintenance.

5) Response set for class (RFC): -

The RFC [6] is a set of methods that can be potentially executed in a response to a message received by an object of that class.

$$RFC = |RS|$$
$$RS = M \cup R$$
$$R = \bigcup_{i=1}^{|M|} R_i$$

where,

M is the set of methods in a given class,

Ri is the set of remote methods directly called from method Mi.

IMPLEMENTATION:

The logic behind implementing the RFC is firstly by calculating the total number of methods and then methods that are invoked from these methods are counted. In understand, RFC cannot be calculated thus we have calculated RFC only using CodeMR tool. Thus, RFC is implemented only at class level. The higher value of RFC indicates more understandability.

6) Lack of Cohesion in method (LCOM): -

In original incarnation of C&K metrics, defined LCOM [6] based on the numbers of pairs of methods that shared references to instance variables. Let Ii be set of instance variables accessed by method Mi. For all possible pairs of methods (Mi, Mj) compute the intersection of Ii ∩ Ij.

- $P = \{(M_i, M_j) \mid I_i \cap I_j = \emptyset\}$
- $Q = \{(M_i, M_j) \mid I_i \cap I_j \neq \emptyset\}$
- $LCOM = \begin{cases} |P| - |Q|, & if |P| > |Q| \\ 0, & otherwise \end{cases}$

IMPLEMENTATION:

The logic behind implementing LCOM would be that, for each class we get the list of all the methods, the size of the list is L, then we have L(L-1)/2 pairs of methods. Then for each method, we have the set called attributes, which stores all the class names of fields the method accessed,

and of methods, the method calls. For each of the other methods, we use a set called attributes that stores the same data corresponding to the that class, then if the intersection of the two sets is not empty, these two methods share attributes and we put the pair into a set Q. After checking with all the methods, the LCOM of the class equals to the total number of method pairs minus 2 times the size of Q. In understand LCOM is implemented as PercentLackOfCohesion.

7) Number of methods (NOM) [5]: -

It will count all the methods defined in a class. Its design property is complexity.

IMPLEMENTATION:

We have implemented the NOM using CodeMR tool. This tool calculates values at class level.

8) Cohesion among methods of class (CAMC) [5]**:**

It computes the relatedness among methods of a class based upon the parameter list of methods.

IMPLEMENTATION:

We have implemented CAMC using CodeMR tool as LCAM (Lack of Cohesion among methods). It is implemented at class level.

9) Line of code (LOC): -

This metric represents the number of lines in the file including comments, empty lines, and actual code.

LOC = # lines in file/class

IMPLEMENTATION:

We have calculated line of code using understand and CodeMR tools. In understand it is implemented as CountLineCode while in CodeMR as LOC.

10) Number of Classes (#C): -

#C represents the total number of classes that are present in a system.

IMPLEMENTATION:

We have computed the values of #C for each project using CodeMR tool.

11) Maintainability Index (MI): -

Maintainability index is a software metric which measures how maintainable (easy to support and change) the source code is. The maintainability index is calculated as a factored formula consisting of lines of code (LOC), cyclomatic complexity and halstead volume [15].

IMPLEMENTATION:

The Maintainability Index (MI) [16] is a set of polynomial metrics, using Halstead's effort and McCabe's cyclomatic complexity, plus some other factors relating to the number of lines of code (or statements in the case of JHawk) and the percentage of comments. The MI is used primarily to determine if code has a high, medium or low degree of difficulty to maintain. It is language independent and was validated in the field by Hewlett-Packard (HP). HP concluded that modules with a MI less than 65 are difficult to maintain, modules between 65 and 85 have reasonable maintainability and those with MI above 85 have excellent maintainability. MI can be calculated at method, class, and package and system level.

IV. ANALYSIS METHOD

In this section, we will talk about the statistical tests, which we performed concerning hypotheses. Based on the collected data and our hypotheses, we conducted regression to find the relation between independent variables (C&K metrics and QMOOD metrics) and dependent variable (maintainability index).

Regression analysis is used as a method of figuring out which of the independent variables are related to the dependent ones and finding the form of relationship between them. As our dependent variable is not categorical thus, we did not perform logistic regression. Therefore, we planned to perform linear regression to predict relationship between the independent variable and dependent variable by fitting a linear equation of a data range as we have discrete and continuous variable. Linear regression finds a best-fit line y=ax+b within the range of data. Linear regression tells us how to examine how the typical value of the dependent variable changes when any one of the independent variables is varied.

For using linear regression, we should know that our model shows normal distribution. We visually checked for normal distribution using the Normal Q-Q plot in R Studio. Q-Q plot (or quantile-quantile plot) draws the correlation between a given sample and the normal distribution. A 45-degree reference line is plotted. If approximately all points fall on the reference line, we can assume that there is normal distribution.

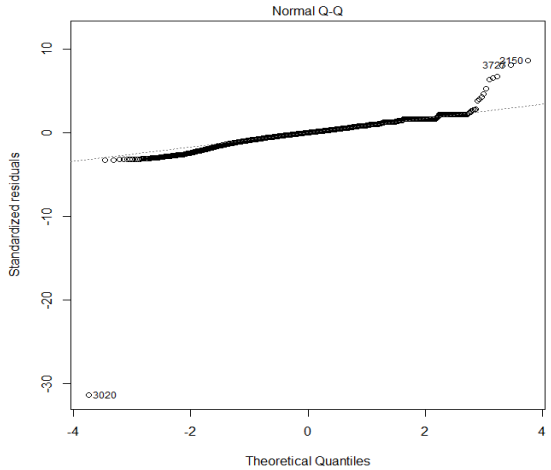

FIGURE 1: Normal Q-Q plot

Thus, from plot we concluded that there is normal distribution and we have modelled our data with linear regression.

To perform linear regression, we need only those variables that are strongly correlated with the dependent variable. Hence, we first calculated the correlation coefficient that measures the extent to which two variables tend to change together and it describes both the strength and the direction of the relationship. In our case, the variables tend to change together but not necessarily at constant rate. Hence, we have chosen spearman coefficient, as we want to evaluate monotonic relationship between the variable.

We used spearman correlation to evaluate the monotonic relationship between the variable as it is based on the ranked values for each variable rather than raw data.

For obtaining results to address our hypotheses, we performed the analysis using the following steps:

1) For each project, we first obtained the correlation co-efficient of each predictor variable (CBO, RFC, DIT, NOC, WMC, LCOM, NOM, LCAM, and LOC) with response variable (maintainability index).
2) Now for fitting the model using linear regression we have used only those metrics that have strong positive/negative correlation with maintainability index. We later also checked for the p-values to confirm their significance with the model.
3) We choose to create the model with the correlated metrics only after comparing the results with other models, which includes those metrics also, that had small positive/negative correlation coefficient with maintainability index. The comparison is done using the Anova test (Chi square test).

V. STUDY RESULTS

As mentioned in step 1 the results of the spearman correlation of each independent variable with maintainability index is given in TABLE 1 that we have discussed in next section.

In TABLE 2, the results of linear regression of maintainability index with strongly correlated independent variables only for each project is shown. The values in results are R-values.

R-values	Software Metrics				
Projects	LOC	LCOM	WMC	LCAM	RFC
Eclipse (2017)	0.1533	0.195	0.1394	0.1707	0.1613
Eclipse (2018)	0.1506	0.1946	0.1383	0.1706	0.1599
Eclipse (2019)	0.1529	0.1943	0.1417	0.1702	0.1599
IntelliJ (2017)	0.2268	0.09493	0.1867	0.1219	0.1534
IntelliJ (2018)	0.213	0.1	0.1735	0.1351	0.1359
IntelliJ (2019)	0.2005	0.09814	0.1575	0.1308	0.1319
NetBeans(2017)	0.2254	0.2006	0.1253	0.26	0.1577
NetBeans(2018)	0.2095	0.2097	0.1154	0.2702	0.1578
NetBeans(2019)	0.1321	0.1083	0.1008	0.1929	0.1579

TABLE 2: R-squared values at class level

As mentioned in step 2, we have fitted a model with all the independent variables (LOC, LCOM, LCAM, WMC, and RFC) to maintainability index. The result of the model i.e. R-values for each project is shown in TABLE 3.

```
20 model3 <- lm(MI~LOC+LCOM+WMC+LCAM+RFC+WMC:LOC+RFC:CBO+LCAM:RFC, data=version1)
21 par(mfrow=c(1,2))
22 plot(model3)
```

PROJECTS	R - VALUES
Eclipse (2017)	0.5202
Eclipse (2018)	0.5187
Eclipse (2019)	0.5189
IntelliJ (2017)	0.5385
IntelliJ (2018)	0.5244
IntelliJ (2019)	0.5032
NetBeans (2017)	0.5911
NetBeans (2018)	0.5849
NetBeans (2019)	0.4786

TABLE 3: R-values for whole model

PROJECTS	Software Metrics									
	CBO	RFC	DIT	NOC	WMC	NOM	LCOM	LCAM	#C	LOC
Eclipse (2017)	-0.37	-0.46	0.02	0.27	-0.82	-0.36	-0.49	-0.45	-0.32	-1
Eclipse (2018)	-0.37	-0.46	0.02	0.27	-0.82	-0.36	-0.49	-0.45	-0.32	-1
Eclipse (2019)	-0.37	-0.46	0.02	0.27	-0.82	-0.37	-0.49	-0.45	-0.32	-1
IntelliJ (2017)	-0.37	-0.46	0.02	0.27	-0.82	-0.36	-0.49	-0.45	-0.32	-1
IntelliJ (2018)	-0.37	-0.46	0.02	0.27	-0.82	-0.36	-0.49	-0.45	-0.32	-1
IntelliJ (2019)	-0.37	-0.46	0.02	0.27	-0.82	-0.36	-0.49	-0.45	-0.32	-1
NetBeans(2017)	-0.45	-0.43	-0.05	0.01	-0.82	-0.31	-0.48	-0.53	-0.29	-1
NetBeans(2018)	-0.46	-0.45	-0.14	0.09	-0.82	-0.33	-0.48	-0.53	-0.28	-1
NetBeans(2019)	-0.44	-0.48	-0.14	0.2	-0.87	-0.29	-0.37	-0.48	-0.3	-1

TABLE 1: Correlation co-efficient values with spearman at class level

After performing linear regression on our model for each project, we have shown the summary of model in tables below. In addition, we have also plotted the model. The plot is MI vs independent variable.

For Eclipse project:

In table 4, 5, 6, there is summary of the model for 2017, 2018, 2019 releases while, figure 2, 3, 4 is the plot.

Eclipse -2017

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	8.500e+01	4.188e-01	202.929	< 2e-16
LOC	-2.516e-02	1.999e-03	-12.586	< 2e-16
LCOM	-1.628e+01	8.197e-01	-19.864	< 2e-16
WMC	-8.167e-02	1.039e-02	-7.861	4.56e-15
LCAM	-1.753e+01	1.294e+00	-13.544	< 2e-16
RFC	-1.163e-01	1.786e-02	-6.512	8.06e-11
LOC: WMC	8.319e-06	2.476e-07	33.595	< 2e-16
RFC: CBO	2.016e-03	1.792e-04	11.251	< 2e-16
LCAM: RFC	-9.478e-02	2.926e-02	-3.239	0.00121

TABLE 4: Summary of model for Eclipse-2017

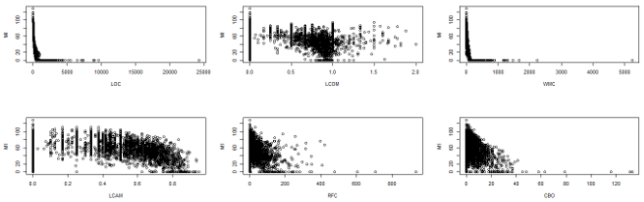

FIGURE 2: Plot of MI vs independent variables for Eclipse-2017

Eclipse -2018

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	8.502e+01	4.220e-01	201.444	< 2e-16
LOC	-2.416e-02	1.986e-03	-12.161	< 2e-16
LCOM	-1.634e+01	8.236e-01	-19.842	< 2e-16
WMC	-8.498e-02	1.030e-02	-8.254	< 2e-16
LCAM	-1.758e+01	1.299e+00	-13.533	< 2e-16
RFC	-1.131e-01	1.728e-02	-6.544	6.53e-11
LOC: WMC	8.205e-06	2.449e-07	33.506	< 2e-16
RFC: CBO	1.986e-03	1.759e-04	11.285	< 2e-16
LCAM: RFC	-9.440e-02	2.855e-02	-3.306	0.000952

TABLE 5: Summary of model for Eclipse-2018

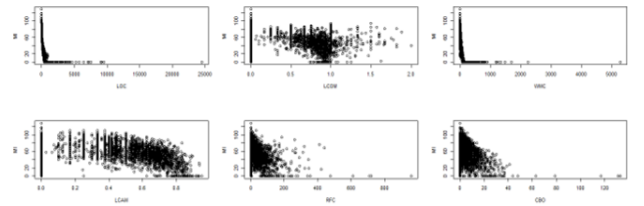

FIGURE 3: Plot of MI vs independent variables for Eclipse-2018

Eclipse -2019

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	8.513e+01	4.193e-01	203.043	< 2e-16
LOC	-2.414e-02	2.004e-03	-12.045	< 2e-16
LCOM	-1.638e+01	8.233e-01	-19.895	< 2e-16
WMC	-8.706e-02	1.031e-02	-8.440	< 2e-16
LCAM	-1.757e+01	1.291e+00	-13.607	< 2e-16
RFC	-1.140e-01	1.702e-02	-6.697	2.34e-11
LOC: WMC	8.415e-06	2.494e-07	33.743	< 2e-16
RFC: CBO	1.994e-03	1.754e-04	11.366	< 2e-16
LCAM: RFC	-9.112e-02	2.803e-02	-3.251	0.00116

TABLE 6: Summary of model for Eclipse-2019

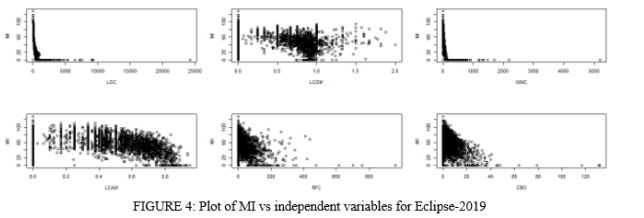

FIGURE 4: Plot of MI vs independent variables for Eclipse-2019

For IntelliJ project:

In table 7, 8, 9, there is summary of the model for 2017, 2018, 2019 releases while, figure 5, 6, 7 is the plot.

Intellij-2017

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	8.259e+01	3.547e-01	232.822	< 2e-16
LOC	-5.846e-02	1.616e-03	-36.165	< 2e-16
LCOM	-9.615e+00	8.660e-01	-11.103	< 2e-16
WMC	-4.818e-02	8.507e-03	-5.664	1.55e-08
LCAM	-1.865e+01	1.177e+00	-15.844	< 2e-16
RFC	-2.483e-01	2.528e-02	-9.823	< 2e-16
LOC: WMC	3.676e-05	9.193e-07	39.989	< 2e-16
RFC: CBO	1.409e-03	3.040e-04	4.635	3.65e-06
LCAM: RFC	1.477e-01	4.044e-02	3.654	0.000261

TABLE 7: Summary of model for Intellij-2017

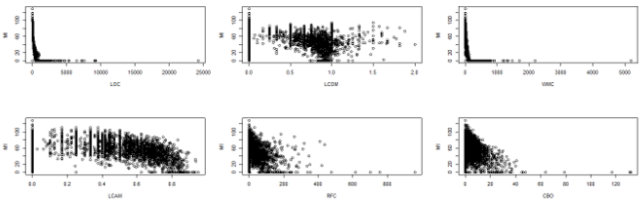

FIGURE 5: Plot of MI vs independent variables for Intellij-2017

Intellij-2018

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	8.274e+01	3.468e-01	238.600	< 2e-16
LOC	-5.307e-02	1.536e-03	-34.561	< 2e-16
LCOM	-1.061e+01	8.644e-01	-12.275	< 2e-16
WMC	-6.298e-02	8.318e-03	-7.572	4.24e-14
LCAM	-2.041e+01	1.131e+00	-18.051	< 2e-16
RFC	-1.936e-01	2.176e-02	-8.898	< 2e-16
LOC: WMC	3.249e-05	8.033e-07	40.444	< 2e-16
RFC: CBO	1.104e-03	2.512e-04	4.397	1.11e-05
LCAM: RFC	1.188e-01	3.445e-02	3.448	0.000568

TABLE 8: Summary of model for Intellij-2018

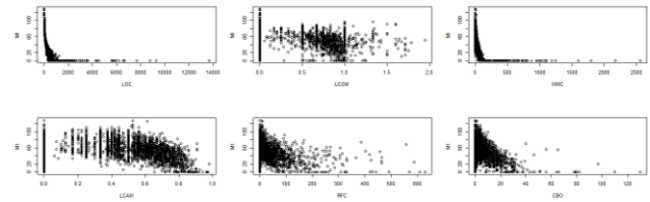

FIGURE 6: Plot of MI vs independent variables for Intellij-2018

Intellij-2019

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	8.161e+01	3.484e-01	234.250	< 2e-16
LOC	-4.574e-02	1.420e-03	-32.205	< 2e-16
LCOM	-1.107e+01	8.698e-01	-12.729	< 2e-16
WMC	-6.696e-02	7.939e-03	-8.434	< 2e-16
LCAM	-1.985e+01	1.124e+00	-17.667	< 2e-16
RFC	-1.490e-01	1.862e-02	-8.004	1.43e-15
LOC: WMC	2.494e-05	6.468e-07	38.562	< 2e-16
RFC: CBO	1.070e-03	2.166e-04	4.941	7.96e-07
LCAM: RFC	6.278e-02	2.943e-02	2.133	0.033

TABLE 9: Summary of model for Intellij-2019

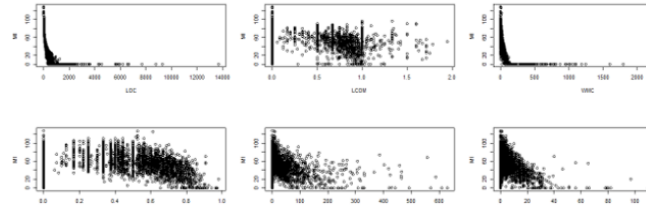

FIGURE 7: Plot of MI vs independent variables for Intellij-2019

For NetBeans project:

In table 10, 11, 12, there is summary of the model for 2017, 2018, 2019 releases while, figure 8, 9, 10 is the plot.

Netbeans-2017

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	8.788e+01	6.468e-01	135.873	< 2e-16
LOC	-3.706e-02	1.770e-03	-20.938	< 2e-16
LCOM	-1.136e+01	1.581e+00	-7.189	8.92e-13
WMC	-6.259e-02	1.058e-02	-5.915	3.84e-09
LCAM	-3.758e+01	2.358e+00	-15.936	< 2e-16
RFC	-2.767e-01	3.846e-02	-7.196	8.46e-13
LOC: WMC	1.657e-05	8.800e-07	18.833	< 2e-16
RFC: CBO	6.868e-04	2.255e-04	3.046	0.00235
LCAM: RFC	2.873e-01	5.378e-02	5.343	1.01e-07

TABLE 10: Summary of model for NetBeans-2017

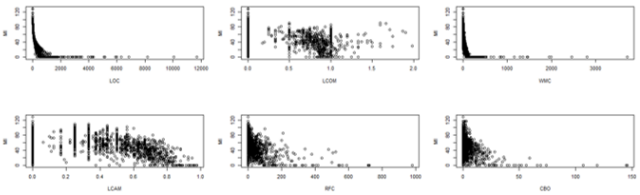

FIGURE 8: Plot of MI vs independent variables for NetBeans-2017

Netbeans-2018

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	8.759e+01	6.512e-01	134.511	< 2e-16
LOC	-3.521e-02	1.686e-03	-20.887	< 2e-16
LCOM	-1.219e+01	1.575e+00	-7.743	1.48e-14
WMC	-5.453e-02	1.015e-02	-5.370	8.71e-08
LCAM	-3.717e+01	2.362e+00	-15.735	< 2e-16
RFC	-2.043e-01	3.727e-02	-5.483	4.68e-08
LOC: WMC	1.451e-05	8.058e-07	18.007	< 2e-16
RFC: CBO	8.267e-04	2.051e-04	4.031	5.75e-05
LCAM: RFC	1.824e-01	5.282e-02	3.453	0.000566

TABLE 11: Summary of model for NetBeans-2018

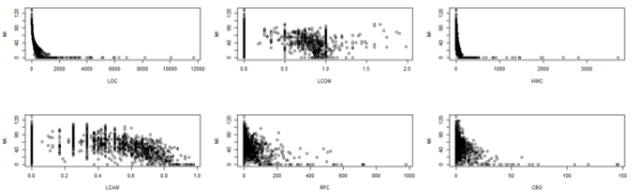

FIGURE 9: Plot of MI vs independent variables for NetBeans-2018

Netbeans-2019

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	8.406e+01	4.155e-01	202.322	< 2e-16
LOC	-2.369e-02	8.935e-04	-26.509	< 2e-16
LCOM	-9.271e+00	7.309e-01	-12.684	< 2e-16
WMC	-3.079e-02	3.747e-03	-8.216	2.45e-16
LCAM	-3.183e+01	1.192e+00	-26.705	< 2e-16
RFC	-1.125e-01	1.049e-02	-10.722	< 2e-16
LOC: WMC	3.926e-06	1.012e-07	38.795	< 2e-16
RFC: CBO	8.128e-04	1.059e-04	7.676	1.84e-14
LCAM: RFC	4.911e-02	1.533e-02	3.203	0.00137

TABLE 12: Summary of model for NetBeans-2019

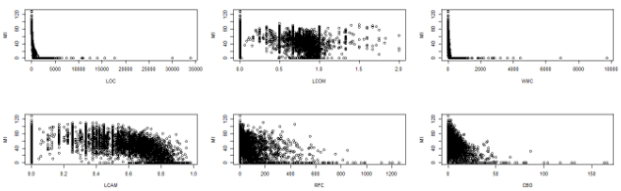

FIGURE 10: Plot of MI vs independent variables for NetBeans-2019

VI. DISCUSSION

In this section, we will discuss about the study results and concluding at last.

From TABLE 1, we have noticed that not all the OO metrics that we use have strong correlation with the maintainability index. Correlation co-efficient tells us that how much both the variables tend to change together. We have calculated the correlation coefficient using spearman co-efficient.

The interpretation of correlation coefficient is:

Correlation	Negative	Positive
None	-0.09 to 0.0	0.0 to 0.09
Small	-0.3 to -0.1	0.1 to 0.3
Medium	-0.5 to -0.3	0.3 to 0.5
Strong	-1.0 to -0.5	0.5 to 1.0

TABLE 13: Interpretation of correlation values

For table 1, using this interpretation we can say that WMC, LCOM, RFC, LCAM, and LOC have strong negative co-relation with maintainability index. While other metrics does not have strong positive/negative correlation with maintainability index.

```
15
16  model2 <- lm(MI~LOC+LCOM+WMC+LCAM+RFC, data = version1)
17  summary(model2)
18  model4 <- lm(MI~LCOM+WMC+LCAM+RFC, data = version1)
19  summary(model4)
20  model3 <- lm(MI~LCOM+WMC+LCAM+RFC+WMC:LOC+RFC:CBO+LCAM:RFC, data
21  #par(mfrow=c(3,3))
22  #plot(MI ~ LOC + LCOM + WMC + LCAM + RFC + WMC:LOC + RFC:CBO + L
23  #abline(model3)
24  summary(model3)
25
26  anova(model3 ,model1, test='Chi')
27  anova(model3, model2, test='Chi')
28
29
27:34  (Top Level)

Console C:/Users/Raj/Desktop/Jem/
> anova(model3, model2, test='Chi')
Analysis of Variance Table

Model 1: MI ~ LCOM + WMC + LCAM + RFC + WMC:LOC + RFC:CBO + LCAM:RFC
Model 2: MI ~ LOC + LCOM + WMC + LCAM + RFC
  Res.Df     RSS Df Sum of Sq  Pr(>Chi)
1   7861 4041300
2   7863 4434733 -2   -393434 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

FIGURE 11: Results of Anova test

Hence, we can say that for fitting a linear model the above-mentioned five variables are best suitable that we have also compared with other model which includes all the variables as discussed in step 3 of analysis. Because sometimes variable that are not correlated can also fit the model. This comparison is done using Anova test (Analysis of variance) as shown in figure 11 which is an analytical test to measure whether observed changes have a significance or not. The strength of each correlated independent variable with dependent variable is as follows:

• Line of Code (LOC):
For calculating maintainability index, LOC is required negatively, which means that lesser the value of LOC more maintainable the software/class is.
• Weighted Method per Class (WMC):
WMC tells us about the cyclomatic complexity for a class, which means that for a class being a more maintainable WMC has to be less.
• Response for Class (RFC):
RFC relates to the methods of a class. A class that has more methods can have a higher RFC. That is related to the size of a class which we can see (from values of table 1 and figure 1) to be related to the number of changes.
• Lack of Cohesion of Methods (LCOM):
A class with high lack of cohesion can have more code than is necessary. Classes would need better structure to be with higher cohesion. This means that the number of changes can be higher. Classes that do more than it are supposed to change more.
• Lack of Cohesion among Methods (LCAM):
A method with high lack of cohesion can have more code than is necessary. Methods would need better structure to be with higher cohesion. This means that the number of changes can be higher. Methods that do more than it are supposed to change more.

TABLE 2 is the result of linear regression performed on all the correlated metrics found from table 1. The values are the R-values computed between dependent variable and each correlated independent variable. R-squared (R2) is a statistical measure that represents the proportion of the variance for a dependent variable that explains by an independent variable or variables in a regression model. R-squared explains to what extent the variance of one variable explains the variance of the second variable.

Table 3 depicts the results obtained when all the correlated independent variables are combined together to form a model. The model is shown in section V. In model, we have added all the correlated variables along with the best internal interactions between them. The internal interactions are decided based on correlation coefficient. The correlation coefficient of those two variables have strong relation are chosen. For ex: RFC, LCAM have 0.8 correlation coefficient, which means that they vary together. Thus adding their interaction increases the R-value of the model. In table 3, the R-values tells us the variance in maintainability index for each release of Eclipse, IntelliJ, and NetBeans. From these R-values, we can see the correlation between the actual and predicted value. For NetBeans 2017 release, R-value is 0.5911 that means there is variance of 59.11% for maintainability index.

In next part, we will discuss about the p-values and the observation of linear regression for each predictor variable for each project.
The p-value is a number between 0 and 1 and interpreted in the following:
• A small p-value (typically $\leqslant$ 0.05) indicates strong evidence against the null hypothesis, meaning that predictor variable is significant to response variable.
•A large p-value (> 0.05) indicates weak evidence against the null hypothesis, so you fail to reject the null hypothesis. Thus, those variables does not have significance.
•P-values very close to the cutoff (0.05) are considered to be marginal (could go either way).

ECLIPSE:
The results we got from linear regression are shown in section V.
From the table 4, 5, 6 it is evident that all the independent variables are significant to maintainability index since all of theirs p-values are less than 0.05. Also from table 2, we can conclude the change in response variable for each predictor variable as follows:
• *LOC:* For Eclipse-2017, we can say that LOC account for 15.33% of variance in maintainability index. Similarly, for Eclipse-2018 and Eclipse-2019 we conclude that LOC account for 15.06% and 15.29% variance respectively.
• *LCOM:* For Eclipse-2017, we can say that the equation related to 19.5%, which specifies for every additional number in LCOM we can expect an increase in the maintainability index by an average of 0.195*LCOM. Similarly, for Eclipse-2018 and Eclipse-2019 we conclude that the equation relates to 19.46% and 19.43%, which specifies for every additional number in LCOM we can expect an increase in the maintainability index by an average of 0.1946*LCOM and 0.1943*LCOM respectively.
• *WMC:* For Eclipse-2017, we can say that WMC indicates for 13.94% of variability of maintainability index. Similarly, for Eclipse-2018 and Eclipse-2019, we conclude that WMC account for 13.83% and 14.17% variability respectively.
• *LCAM:* For Eclipse-2017, we can say that the equation related to 17.07%, which specifies for every additional number in LCAM we can expect an increase in the maintainability index by an average of 0.1707*LCOM. Similarly for Eclipse-2018 and Eclipse-2019, we conclude that the equation relates to 17.06% and 17.02%, which specifies for every additional number in LCAM we can expect an increase in the maintainability index by an average of 0.1706*LCOM and 0.1702*LCOM respectively.

- *RFC:* For Eclipse-2017, we can say that RFC account for 16.13% of variance in maintainability index. Similarly, for Eclipse-2018 and Eclipse-2019 we conclude that RFC account for 15.99% and 15.99% variance respectively.

INTELLIJ:
The results for all the releases of IntelliJ from linear regression are shown as below:

- *LOC:* For IntelliJ-2017, we can say that the equation related to 22.68%, which specifies for every additional number in LOC we can expect an increase in the maintainability index by an average of 0.2268*LOC. Similarly for IntelliJ-2018 and Intellij-2019 we conclude that the equation relates to 21.3% and 20.05%, which specifies for every additional number in LOC we can expect an increase in the maintainability index by an average of 0.213*LOC and 0.2005*LOC respectively.

- *LCOM:* For IntelliJ-2017, we can say that the equation related to 9.44%, which specifies for every additional number in LCOM we can expect an increase in the maintainability index by an average of 0.0944*LCOM. Similarly for IntelliJ-2018 and Intellij-2019 we conclude that the equation relates to 10.00% and 9.81%, which specifies for every additional number in LCOM we can expect an increase in the maintainability index by an average of 0.1*LCOM and 0.0981*LCOM respectively.

- *WMC:* For IntelliJ-2017, we can say that the equation related to 18.67%, which specifies for every additional number in WMC we can expect an increase in the maintainability index by an average of 0.1867*WMC. Similarly for IntelliJ-2018 and Intellij-2019 we conclude that the equation relates to 17.35% and 15.75%, which specifies for every additional number in WMC we can expect an increase in the maintainability index by an average of 0.1735*WMC and 0.1575*WMC respectively.

- *LCAM:* For IntelliJ-2017, we can say that the equation related to 12.19%, which specifies for every additional number in LCAM we can expect an increase in the maintainability index by an average of 0.1219*LCAM. Similarly for IntelliJ-2018 and Intellij-2019 we conclude that the equation relates to 13.51% and 13.08%, which specifies for every additional number in WMC we can expect an increase in the maintainability index by an average of 0.1351*LCAM and 0.1308*LCAM respectively.

- *RFC:* For IntelliJ-2017, we can say that the equation related to 15.34%, which specifies for every additional number in RFC we can expect an increase in the maintainability index by an average of 0.1534*RFC. Similarly for IntelliJ-2018 and Intellij-2019 we conclude that the equation relates to 13.59% and 13.19%, which specifies for every additional number in RFC we can expect an increase in the maintainability index by an average of 0.1359*RFC and 0.1319*RFC respectively.

NETBEANS:
The results for all the releases of NetBeans from linear regression are shown as below:

- *LOC:* For NetBeans-2017, we can say that LOC indicates for 22.54% of variability of maintainability index. Similarly for Eclipse-2018 and Eclipse-2019, we conclude that LOC account for 20.95% and 13.21% variability respectively.
- *LCOM:* For NetBeans-2017, we can say that LCOM indicates for 20.06% of variability of maintainability index. Similarly for Eclipse-2018 and Eclipse-2019, we conclude that LCOM account for 20.97% and 10.83% variability respectively.
- *WMC:* For NetBeans-2017, we can say that WMC indicates for 12.53% of variability of maintainability index. Similarly for Eclipse-2018 and Eclipse-2019, we conclude that WMC account for 11.54% and 10.08% variability respectively.
- *LCAM:* For NetBeans-2017, we can say that LCAM indicates for 26% of variability of maintainability index. Similarly for Eclipse-2018 and Eclipse-2019, we conclude that LCAM account for 27.02% and 19.29% variability respectively.
- *RFC:* For NetBeans-2017, we can say that RFC indicates for 15.77% of variability of maintainability index. Similarly for Eclipse-2018 and Eclipse-2019, we conclude that RFC account for 15.78% and 15.79% variability respectively.

VII. THREATS TO VALIDITY

Construct validity:
We used different projects with many versions for each and we computed class level metrics, however the metric we used to compute the change proneness, maintainability index may be insufficient. There can be other techniques for computing number of changes, for example number of commits, which may be more effective but since it was difficult for us to get the value for all the systems, hence we have to limit it to maintainability index.

Internal validity:
The data was collected data for each system from their version repositories. In the data extraction, we included only the production code of each system and ignored all folders that were not directly related to the functionality of the system like test folders.

External validity:
This study is limited to Java projects only. Also, the selection of different release versions for the systems play major roles. Some systems like eclipse have very frequent releases while NetBeans and IntelliJ does not.

VIII. CONCLUSIONS

Based on our observations and exploratory study, we concluded that WMC, RFC and LCOM from CK metrics have negative relation with the maintainability index (Change Proneness) meaning that they accept the alternate hypotheses. Since complexity is related to WMC and with more complex code, it becomes difficult to maintain. Higher value of RFC indicates more faults as it is related to method calls in response to message received by the object of that class. Also, higher value of LCOM indicates class needs to be separated and making the class difficult to maintain. While DIT, NOC and CBO do not have any relation with the maintainability index means they accept null hypotheses. Similarly, LOC and LCAM from QMOOD metrics have negative relationship with the maintainability index (Change Proneness) meaning that they reject null hypotheses.

Using the data, we gathered, we observed that using DIT, NOC and CBO we didn't found strong correlation and there was a drop in prediction power while predicting the model. Furthermore, DIT and NOC are more related to the inheritance and maintainability index does not count for CBO, hence there is no relation with the change proneness.

Adding further, the quality attributes of QMOOD models: understandability, functionality, reusability, and extendibility are also in turn affected. Because, they are directly related to cohesion (LCAM), complexity (LOC).

To reconsolidate, this empirical validation provides practitioners with some empirical evidence demonstrating that WMC, RFC, LCOM, and LCAM metrics can be helpful in predicting the change proneness. While LOC is not much helpful as it is not much controllable.

IX. REFERENCES

[1] Massimiliano Di Penta1, Luigi Cerulo1, Yann-Ga ¨ el Gu ´eh ´eneuc2, and Giuliano Antoniol3 "An Empirical Study of the Relationships between Design Pattern Roles and Class Change Proneness"

[2] Kazuhiro Yamashita, Changyun Huang, Meiyappan Nagappan, Yasutaka Kamei, Audris Mockus, Ahmed E. Hassan, and Naoyasu Ubayashi, "Thresholds for Size and Complexity Metrics: A Case Study from the Perspective of Defect Density" 2016.

[3] A Review of Studies on Change Proneness Prediction in Object Oriented Software, international Journal of Computer Applications (0975 – 8887) Volume 105 – No. 3, November 2014.

[4] Understanding change-proneness in OO software through visualization, 11th IEEE International Workshop on Program Comprehension, 2003.

[5] QMOOD metric sets to assess quality of Java program, 2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT).

[6] Investigating the OO characteristics of software using CKJM metrics, 2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions)

[7] V. R. Basili and B. T. Perricone. Software errors and complexity: An empirical investigation. ACM Commun., 27(1):42–52, 1984.

[8] D. Coleman, B. Lowther, and P. Oman. The application of software maintainability models in industrial software systems. J. Syst. Softw., 29(1):3 16, 1995.

[9] B. T. Compton and C. Withrow. Prediction and control of software defects. J. Syst. Softw. ,12(3):199–207, 1990.

[10] T. McCabe. A complexity measures. IEEE Trans. Softw. Eng., SE-2(4):308 – 320, 1976.

[11] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects," in Proceedings of the 11th Working Conference on Mining Software Repositories, Hyderabad, India, 2014, pp. 192-201.

[12] Nikolaos Tsantalis, Alexander Chatzigeorgiou and George Stephanides, , IEEE Transactions on Software Engineering, The Probability of Change in Object-Oriented Systems, July 2005 .

[13] Foutse Khomh, Massimiliano Di Penta and Yann-Ga¨el Gu´eh´eneuc, "An Exploratory Study of the Impact of Code Smells on Software Change-proneness", 2009.

[14] Amandeep Narula, Ankit Chitlangia, Arash Abolfathi, Armaan Gill, Golnoush Lotfi, Guilherma Padua, Jitin Sharda, Marzieh Monrazeri, Tarnum Sharma, Vikram Sarangan, "C&K Metric Suite: an indicator of change proneness" .

[15] http://www.projectcodemeter.com/cost_estimation/help/GL_maintainability.htm

[16] http://www.virtualmachinery.com/jhawkmetricsmethod.htm