International Journal of Rough Sets and Data Analysis
Volume 6 • Issue 3 • July-September 2019

Analyzing Evolution Patterns of Object-Oriented Metrics:
A Case Study on Android Software

Ruchika Malhotra, Delhi Technological University, New Delhi, India
Megha Khanna, Sri Guru Gobind Singh College of Commerce, University of Delhi, New Delhi, India

ABSTRACT

Software evolution is mandatory to keep it useful and functional. However, the quality of the evolving software may degrade due to improper incorporation of changes. Quality can be monitored by analyzing the trends of software metrics extracted from source code as these metrics represent the structural characteristics of a software such as size, coupling, inheritance etc. An analysis of these metric trends will give insight to software practitioners regarding effects of software evolution on its internal structure. Thus, this study analyzes the trends of 14 object-oriented (OO) metrics in a widely used mobile operating system software, Android. The study groups the OO metrics into four dimensions and analyzes the trends of these metrics over five versions of Android software (4.0.2-4.3.1). The results of the study indicate certain interesting patterns for the evaluated dimensions, which can be helpful to software practitioners for outlining specific maintenance decisions to improve software quality.

KEYWORDS

Android, Cohesion, Coupling, Inheritance, Metric Trends, Object-Oriented Metrics, Size, Software Evolution

1. INTRODUCTION

Software has revolutionized all aspects of our life. However, once operational, it needs constant upgrade and change in order to maintain its usefulness. There could be various reasons for a software to evolve which includes change in requirements, rectification of existing errors or technological advancement in the software's environment (Malhotra and Khanna, 2017). A prime concern of software developers is effective software design so that changes in a software do not result in its poor quality. Thus, we need to constantly monitor software quality, in order to ensure successful software products with satisfied customers.

An efficient method to predict software quality is to use various metrics extracted from source code. These metrics depict various characteristics of a class in an OO software such as its reusability, its dependence on other classes, its cohesiveness, size, etc. The metrics are a representative of a software's internal structure. Though, previous literature studies (Elish and Al-Khiaty, 2013; Lu et al., 2012; Malhotra and Khanna, 2017) have successfully established a number of software quality prediction models with the help of these software metrics, there have been few studies which analyze the trends of these metrics in an evolving software. There is an urgent need to analyze the trends of these metrics as they will be helpful to software practitioners in: a) understanding the effects of evolution on the software's quality; b) assessing a software's internal structure and taking proper corrective actions to avoid its degradation; and c) planning and allocation of proper resources during

DOI: 10.4018/IJRSDA.2019070104

maintenance. Simply, prediction of problematic parts does not lead to software quality improvement. The essence of analyzing metrics values is to outline a systematic plan which strengthens a software's internal structure and prevents its degeneration.

This study assesses the trends depicted by 14 OO metrics in five application packages of a popular mobile operating system (OS), Android. We have chosen Android OS for evaluation as the software is open-source in nature and there has been a steep rise in its demand since its launch. The classes from five application packages of Android software namely Gallery2, Email, Contacts, Calendar and MMS were analyzed over five versions (4.0.2, 4.0.4, 4.1.2, 4.2.2 and 4.3.1). The 14 investigated OO metrics were categorized in four dimensions according the OO characteristic they represent. These dimensions include coupling, cohesion, size and inheritance. For each of the four dimensions, we examined the trends of OO metrics. In order to examine the trends, we extracted the classes which were common to each of the five investigated versions of the software. These common classes were then divided into two categories i.e. changed classes (CC) and unchanged classes (UCC) on the basis of whether a class had undergone change in any of the five investigated versions or not. The characteristics of both categories of classes were examined to ascertain generalized trends. Furthermore, the actually changed classes in each consecutive version were also observed on the basis of change in its metric values along each dimension. The change in metric values were divided into three categories viz. "Constant", "Increasing," and "Decreasing."

The trends observed give insight into software evolution process and can be used by software practitioners for improved software design. These can help in effective maintenance activities and development of better-quality software products.

The current study is organized in nine sections. Section 2 states related literature studies. Section 3 and 4 discusses the various metrics investigated in this work and the hypothesis investigated by the authors with respect to metric trends. Section 5 states the process of collecting metric data from the source code. Section 6 discusses the trends of OO metrics and evaluates the acceptance or rejection of the investigated hypothesis. A comparison of the obtained results with the literature studies is presented in Section 7. The threats of the study are mentioned in Section 8 and the results are summarized in Section 9.

2. RELATED WORK

Various studies in literature have developed successful models to predict software change (Elish and Al-Khiaty, 2013; Lu et al., 2012; Malhotra and Khanna, 2017) and maintainability effort (Fioravanti and Nesi, 2001; Thamburaj and Aloysius, 2017) by using OO metrics as predictors. Although, these studies evaluate the quality of the software on the basis of OO metrics, they do not analyze the trends of these metrics over various versions of a software.

Counsell et al. (2006) interpreted the utility and significance of three cohesion metrics. In order to do so they validated the metrics obtained from three C++ software systems. Nasseri et al. (2008) investigated seven open-source software systems to assess the patterns depicted by four inheritance metrics. Their study concluded that software systems tend to increase in size by adding a greater number of new classes i.e. "breadth-wise" rather than adding classes somewhere in the inheritance hierarchy i.e. "depth-wise". A study by Mubarak et al. (2010) evaluated the relationship between two coupling metrics i.e. "fan-in" and "fan-out" using five open source software systems. They found a correlation between these metrics in majority of the investigated systems.

A study by Lee et al. (2007) rigorously analyzed the evolution of an open source system, JFreeChart. Their study confirmed gradual increase in the number of classes in majority of the releases. However, this increase in number of classes was found correlated with coupling metrics, and not cohesion metrics. A study by Alenezi and Zarour (2015) evaluated the "modularity" evolution of two open-source software systems by analyzing OO metrics corresponding to coupling, cohesion and complexity. They found that the modularity of the investigated systems did not improve with time.

They further extended their study to include two other open-source projects and found similar results with respect to modularity (Alenezi and Zarour, 2016). However, on the other hand, they found that the defect density showed improvement in the investigated systems with time.

A study by Saini et al. (2018) investigated whether there is any difference between the quality of cloned methods and non-cloned methods by validating 3562 open-source projects developed in Java language. The quality was assessed by evaluating the values of 27 software metrics pertaining to the categories of size, documentation and modularity. Their results show that when controlling for size, there was no statistically significant difference between the quality of cloned and non-cloned methods in majority of the observed metrics. Kaur (2015) analyzed the maintenance activity of a reusable software component, which is open-source is nature. They specifically assessed two software metrics namely, Depth of Inheritance Tree (DIT) and Class to Leaf Depth (CLD). They found that during maintenance activity, it is likely that new classes are added at shallow levels, however, classes may be removed from any level in the inheritance hierarchy including deep levels. Singh and Bhattacherjee (2014) assessed four complexity metrics including Weighted Methods per Class (WMC) in 38 versions of JFreeChart. They concluded that increase in complexity decreases the understandability of the software. Alenezi and Almustafa (2015) assessed the complexity evolution for five open-source projects by analyzing line of code (LOC) and cyclomatic complexity (CC) metrics over ten releases of the software systems. They confirmed that software evolution has led to increase in complexity of the observed systems.

As indicated above, there are very few studies which have analyzed metric trends. Thus, it is important to conduct more such studies to validate previous findings and understand the structure of an evolving software. Moreover, majority of these studies have focused on only one dimension i.e. either inheritance or complexity or cohesion or any other. Therefore, an extensive study which covers metrics belonging to multiple dimensions is crucial to understand their effect on each other and on software evolution. The current study evaluates 14 metrics belonging to four different dimensions and uses Android, a popular open-source software for validation. Furthermore, we compare our results with previous studies to strengthen the obtained conclusions.

3. OO METRICS ANALYZED IN THE STUDY

The various metrics investigated in the study corresponding to the dimensions they address are stated in Table 1. These OO metrics have been commonly used by various researchers in the software engineering community. The definitions of these metrics are stated in Table 1 (https://www.spinellis.gr/sw/ckjm/doc/metric.html).

4. HYPOTHESIS

For each of the four dimensions, the authors formulate a set of hypotheses, which are assessed by analyzing the trends of the metrics mentioned in Table 1.

A. Hypothesis for Size Metrics
- *H1 (WMC)*: The number of methods in a class increases as the software system evolves. (*Null Hypothesis:* The number of methods in a class decreases as the software system evolves.)
- *H2 (NPM)*: The number of public methods in a class increases as the software system evolves. (*Null Hypothesis:* The number of public methods in a class decreases as the software system evolves.)
- *H3 (AMC)*: In an evolving software system, there is an increase in average method size of a class.

Table 1. OO metrics

Attribute	Metric with Definition
Size	Weighted Methods per Class (WMC) is the sum of all method complexities. A complexity value of 1 is allocated to each method, thus it is a representative of number of a class's methods (Chidamber and Kemerer, 1994).
	Number of Public Method (NPM) counts all public methods of a class.
	Average Method Complexity (AMC) computes the average number of java binary codes as a representative of method size.
	Lines Of Code (LOC) counts the number of lines in java binary code of a class.
Cohesion	Lack of Cohesion in methods (LCOM) represents a count of pairs of methods of a specific class that do not share any of the class's members and are hence not related (Chidamber and Kemerer, 1994).
	Cohesion among methods (CAM) of a class estimates the connectivity amongst class methods on the basis of their parameter list. A summation of different parameter types used by all methods of a class is divided by the product of total count of methods of a class and the total number of different parameter types.
	LCOM3 (Henderson-Sellers 1998) is computed as defined in Equation 1. $$\left(\frac{1}{v}\sum_{i=1}^{v}\lambda(v_j)\right)-m\Big/(1-m) \quad (1)$$ m: No. of methods, v: No. of attributes; λ(v): No. of methods that access variable v.
Coupling	Coupling between objects (CBO) represents the number of coupled classes to a specific class (Chidamber and Kemerer, 1994). It is the sum of both afferent couplings (Ca), which represents the count of classes using a specific class (fan-in) and efferent coupling (Ce), which represents the classes which are used by a specific class (fan-out) (Martin, 2002). Afferent coupling can also be termed as export coupling and Efferent coupling as import coupling.
	Response For a Class (RFC) estimates the number of methods which respond if a specific class receives a message (Chidamber and Kemerer, 1994).
Inheritance	Depth of Inheritance Tree (DIT) represents the level of the class in the inheritance tree (Chidamber and Kemerer, 1994).
	Number of Children (NOC) counts the number of immediate subclasses (Chidamber and Kemerer, 1994).
	Measure of functional Abstraction (MFA) is computed as the ratio of inherited methods to the total number of accessible methods of a class.

(*Null Hypothesis:* In an evolving software system, the average method size of a class will remain constant.)

- *H4 (LOC)*: The LOC of a class increases as the software system evolves.
  (*Null Hypothesis:* The LOC of a class decreases as the software system evolves.)

B. Hypothesis for Cohesion Metrics

As the software evolves, classes should be designed in a manner to increase their cohesiveness. A lower value of LCOM and LCOM3 indicates better cohesiveness. On the contrary, a higher value of CAM represents a more cohesive class. A strong overlap among attribute types and method parameters is an indication of strong cohesion. If CAM is low, methods may be coupled to each other via attributes resulting in complex class design.

- *H5 (LCOM)*: The LCOM values of a class decrease as the software system evolves.
  (*Null Hypothesis:* The LCOM values of a class increase as the software system evolve.)

- *H6 (LCOM3)*: Variable usage in a class (lower LCOM3 values) improves as the software system evolves.
  (*Null Hypothesis:* The design of the classes deteriorates with respect to variable usage (higher LCOM3 values) as the software system evolves.)
- *H7 (CAM)*: Due to evolution in a software system, cohesion in terms of CAM values increase.
  (*Null Hypothesis:* Due to evolution in a software system, cohesion in terms of CAM values decreases.)

C. Hypothesis for Coupling Metrics

As the software evolves, the classes should be designed in such a manner that their dependency on each other i.e. their coupling decreases.

- *H8 (Ca)*: A class depicts lower export coupling as the software system evolves.
  (*Null Hypothesis*: A class depicts higher export coupling or no change in its export coupling values as the software system evolves.)
- *H9 (Ce)*: A class depicts lower import coupling as the software system evolves.
  (*Null Hypothesis*: A class depicts higher import coupling or no change in its import coupling values as the software system evolves.)
- *H10 (CBO)*: A class becomes loosely coupled (lower import and export coupling) as the software system evolves.
  (Null Hypothesis: A class becomes tightly coupled (higher import and export coupling) or there is no change in its coupling values as the software system evolves.
- *H11 (RFC)*: A class has lower response set in terms of number of methods (decrease in RFC values) as the software system evolves.
  (Null Hypothesis: A class has higher response set in terms of number of methods (increase in RFC values) as the software system evolves.

D. Hypothesis for Inheritance Metrics

- *H12 (DIT)*: The inheritance hierarchy (DIT value) would increase as the software system evolves.
  (Null Hypothesis: The inheritance hierarchy (DIT value) would remain same as the software system evolves.
- *H13 (NOC)*: More subclasses will be extended from previously existing classes (increase in NOC value), as the software system evolves.
  (Null Hypothesis: No new subclasses will be extended from previously existing classes (constant NOC value), as the software system evolves.
- *H14 (MFA)*: A class uses higher number of inherited methods (increase in MFA value) as the software system evolves.
  (Null Hypothesis: The number of inherited methods used by a class remains constant (constant MFA value) as the software system evolves.

5. EMPIRICAL DATA COLLECTION

The study uses Android OS for empirical validation as it is a popular open-source project which encapsulates approximately 80% of the mobile OS market. Five versions (4.0.2, 4.0.4, 4.1.2, 4.2.2, and 4.3.1) of Gallery2, email, contacts, calendar and MMS application packages were collected using defect collection and reporting system (DCRS) tool (Malhotra et al., 2014). The tool computes OO metrics with the aid of CKJM tool (http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/metric.html). The change statistics i.e. number of lines inserted, modified or removed in classes were extracted by assessing the change logs from GIT repository. We first computed change between two consecutive versions of an application package. However, in order to analyze trends, we extracted the classes common to all the five versions of an application package i.e. classes that were existent throughout from version

4.0.2-4.3.1. The number of common classes in each application package and the percentage of changed classes in consecutive versions of each package for the common classes is mentioned in Table 2. The table also states the actual number of common classes ("Actual Changed Class") which changed while progressing from one version to another. For instance, 90 classes changed out of the common classes while progressing from version 4.0.2-4.0.4. It should be noted that the actual changed classes are a cumulative figure computed by analyzing all the five application packages.

Table 2. Dataset details

Application package	No. of common classes	Percentage Change (%)			
		4.0.2-4.0.4	4.0.4-4.1.2	4.1.2-4.2.2	4.2.2- 4.3.1
Gallery2	184	15.2	35.3	34.8	27.7
Email	465	4.1	73.5	65.6	1.5
Contacts	156	5.1	42.3	10.9	8.3
Calendar	74	29.7	60.8	16.2	55.4
MMS	191	6.8	35.6	52.4	7.8
Actual Changed Classes		90	431	496	127

6. METRIC TRENDS

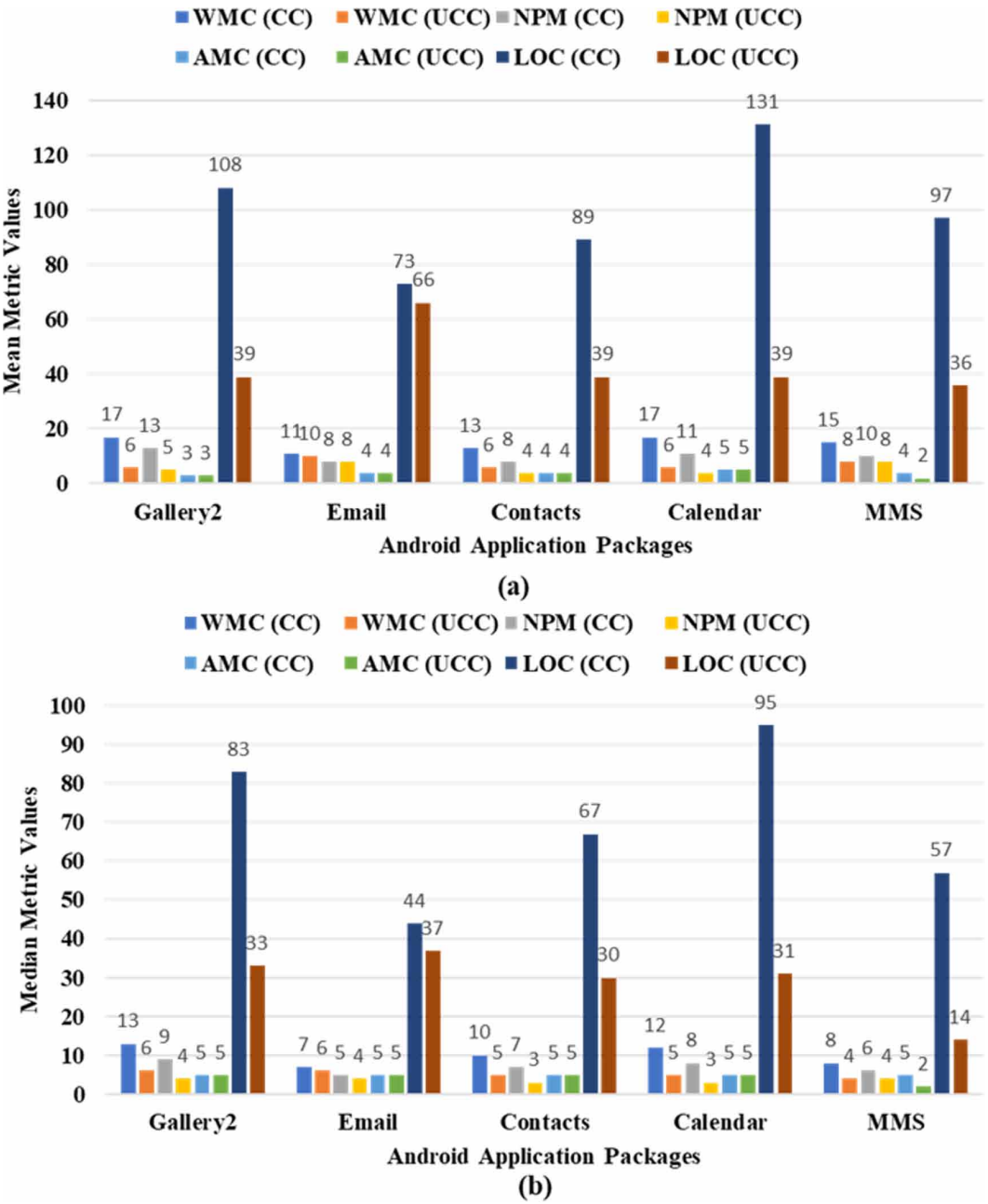
The trends of metrics were analyzed for all the four dimensions for all the investigated application packages of the study. This section states the trends corresponding to each metric dimension. As discussed in Section 1, the common classes were categorized into CC and UCC. Also, those classes which actually changed between two specific consecutive versions were identified as having "Constant", "Increasing" or "Decreasing" trends in consecutive versions. A class is categorized as having a "Constant" trend if the value of a specific metric does not change for the class while transmitting from version A to consecutive version B. A class trend is termed as "Increasing" if the value of a specific metric increased for the class while transmitting from version A to consecutive version B. Similarly, a "Decreasing" trend represents the decrease in metric value as compared to the metric value observed in the version A than the metric value when it was transferred to consecutive version B.

6.1 Trends of Size Metrics

Four metrics corresponding to class size were analyzed namely WMC, NPM, AMC and LOC. Figure 1(a) depicts the mean values of these metrics over all the five versions (4.0.2-4.3.1) for CC and UCC. According to the figure, it may be noted that the CC have higher mean values than UCC. For instance, in Gallery2, the mean NPM values of CC (13) is greater than that of UCC (5). The median values depicted in Figure 1(b) also show a similar trend. An exception to these trends was the E-mail application package. There was not much difference in the mean and median metric values of WMC, NPM and AMC for the package. Moreover, the difference in AMC values was only visible in MMS application package. In all the other application packages, there was not much change in the mean values of the AMC metric for CC and UCC. An analysis of LOC metric values in Figure 1(a) indicates a difference of 10%-70% in the mean LOC values of CC and UCC.

We also observe the trend frequency of the four observed size metrics for all the actual changed classes (refer Table 1) in consecutive versions. The number of classes depicting "constant" (Const.), "increasing" (Inc.) and "decreasing" (Dec.) trends is depicted in Table 3. A prominent trend for the

Figure 1. Mean and median values of size metrics



LOC metric was increase in class size as 47% (4.0.2-4.0.4), 35% (4.0.4-4.1.2) and 41% (4.2.2-4.3.1) of actual changed classes showed an "increasing" trend. Though, only 16% classes showed an "increasing" trend in 4.1.2-4.2.2. This was because in 50 common classes, the added and deleted number of lines were same, leading to no change in class size. The next popular trend for LOC metric was "constant" followed by the "decreasing" trend. An increase in class size, may not always lead to increase in number of class methods (WMC) and in number of public methods (NPM). Thus, most classes depicted a

"constant" trend for WMC and NPM. However, apart from version 4.1.2-4.2.2, 23%-39% of classes depicted an "increasing" trend for these metrics. It may be noted that 97%-99% of classes depicted a constant trend for AMC metric. Thus, according to the above discussion, hypothesis H1, H2 and H4 are accepted and H3 is rejected. The authors accept hypothesis H1 and H2, as "increasing" is the most common trend after "constant" and every change might not result in change of NPM or WMC.

Furthermore, we also conducted Wilcoxon signed rank test at $\alpha = 0.05$ with the mean size metric values i.e. WMC, NPM, AMC and LOC (of all the five application packages) of CC and UCC to ascertain if CC exhibit higher values for size metrics. The p-value of the test was computed as 0.001 indicating significant differences between mean size metric values of CC and UCC. Wilcoxon test results pointed out that larger size metrics are exhibited by CC classes.

Table 3. Version specific size metric trends

Size Metrics	4.0.2-4.0.4			4.0.4-4.1.2			4.1.2-4.2.2			4.2.2-4.3.1		
	Const.	Inc.	Dec.	Const.	Inc.	Dec.	Const.	Inc.	Dec.	Const.	Inc.	Dec.
WMC	51	35	4	266	112	53	417	66	13	70	42	15
NPM	59	28	3	288	100	43	426	54	16	84	33	10
AMC	89	0	1	419	5	7	487	2	7	125	1	1
LOC	40	42	8	217	152	62	399	80	17	54	52	21

6.1.1 Observations Corresponding to Size Metrics

- The mean class size in terms of LOC, NPM and WMC is much higher for CC as compared to UCC. This indicates that classes tend to increase in size in terms of LOC, number of total methods and number of public methods. However, the average method size of each class does not vary much due to evolution. These trends indicate that additional functionality or correction of errors lead to a greater number of lines of code and more number of methods of a class, specifically public methods as there is high difference in mean values of WMC and NPM. However, in general the size of methods of a class do not vary much.
- It was also observed that during evolution, the most common trend was increase in class size. Thus, we observe that in very few cases, the class size decreases during evolution. This may indicate that though new functionality is added in terms of methods or lines of code, it is rare that the evolution leads to decrease in class size.
- Since, in majority of cases the AMC metric exhibited constant trend for changed classes, it may be deciphered that the added lines of code generally lead to creation of new methods. It is rare that the size of an already existing class method would increase.

6.2 Trends of Cohesion Metrics

In order to analyze the cohesive characteristics of the classes, the mean (Figure 2), median (Table 4), minimum and maximum values of three metrics namely LCOM, LCOM3 and CAM were examined over all the five versions of the application packages. A lower value for LCOM and LCOM3 is desired but for CAM a higher value is desired. It was observed from Figure 2(a) that the UCC exhibited better cohesiveness as compared to the CC using LCOM values. As shown in the figure, for calendar application package, the mean value of LCOM for UCC was 24. However, the mean LCOM value for CC was found to be 296. Similarly, as depicted in Table 5, there was a large difference in the median LCOM values of UCC (10) for the Calendar application package as compared to the LCOM

values of CC (66). However, the LCOM metric does not have a fixed scale, as the maximum value that can be attained by the metric is dependent on the number of pairs of methods. Thus, in order to actually conclude that the LCOM value is low and a class is cohesive, we have to take into account the number of method pairs. For instance, an LCOM value of 2 when the number of method pairs is 4 is different from an LCOM value is 2 and the number of method pairs is 50. The class is more cohesive in the latter case. Thus, we analyse the LCOM3 and CAM metrics for a better evaluation of cohesion of classes as the LCOM metric has been criticized in literature (Counsell et al. 2006; Gupta 1997).

The LCOM3 metric computes the cohesiveness of a class by taking into account the effective use of various class variables. According to LCOM3 values depicted in Figure 2(b), the CC obtained a lower mean value as compared to the UCC. For instance, the CC obtained a mean LCOM3 value of 1.25, while the UCC obtained a mean LCOM3 metric of 1.67 for MMS application package. A similar trend observed by analysing the median values (Table 4). This indicates lower cohesiveness of the UCC when evaluated using the LCOM3 metric. On the contrary, the definition of CAM metric indicates computation of a class's cohesiveness by evaluating the types of method parameters. A CAM value nearer to 1 is preferred for a well-designed cohesive class. According to the values depicted in Figure 2(b), the mean values of CAM metric is higher for UCC as compared to the CC. For the Email application package, the CC exhibit a mean CAM value of 0.4 as compared to the mean CAM value of 0.6 for UCC group. A similar trend can be observed from the median values of CAM metric (Table 4). This indicates higher cohesiveness of the UCC, when evaluated by using the CAM metric. These observations lead to a very interesting pattern. The CC are more cohesive than UCC category, when we compute cohesion according to effective use of various class attributes. However, the CC classes exhibit a lower cohesion when the types of parameters used in the method is evaluated. Thus, the CC were better designed with respect to variable usage but poorly designed with respect to types of parameters when compared with the UCC.

We also observed the minimum values obtained by the LCOM3 metric which were all in the range of 1.01-1.07 in all the application packages of the Android software. A value of 1.01 or more indicates extremely deficient cohesive nature of a class. Such classes should be redesigned as it indicates ineffective variable usage. Also, classes were observed with an LCOM3 value of 2. Again, such poorly designed classes should be restructured to improve their cohesiveness.

The version specific trends of cohesion metrics are depicted in Table 5. It may be noted that a change in class may not always lead to change in cohesion metrics. Thus, "constant" trend is observed in a large number of classes. However, we analyze the "increasing" and "decreasing" trends to evaluate the hypothesis and observe how cohesion metrics change with evolution. The prominent trends in cohesion metrics were increase in LCOM values (13%-39%), decrease in LCOM3 values (12%-38%)

Figure 2. Mean values of cohesion metrics

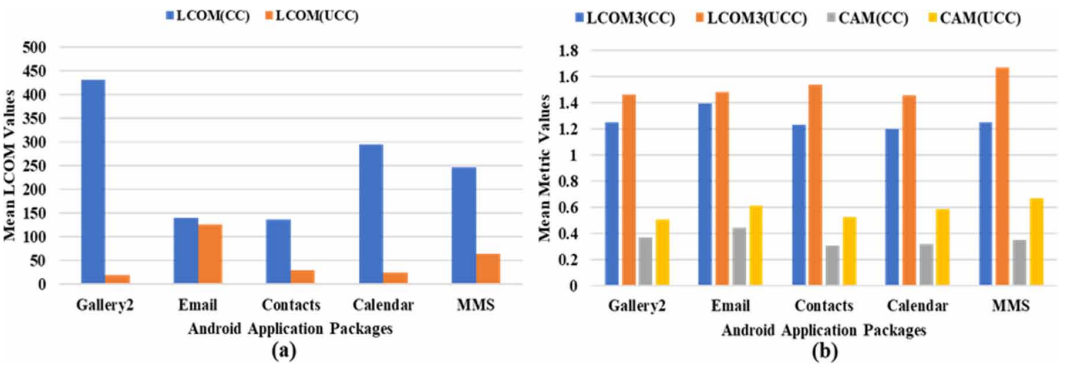


Table 4. Median values of cohesion metrics

Application Package	LCOM (CC)	LCOM (UCC)	LCOM3 (CC)	LCOM3 (UCC)	CAM (CC)	CAM (UCC)
Gallery2	78	15	1.1	1.3333	0.2963	0.4643
Email	21	15	1.2	1.3333	0.4	0.5
Contacts	45	10	1.1111	2	0.2716	0.45
Calendar	66	10	1.0909	1.3333	0.2407	0.48
MMS	28	6	1.1429	2	0.325	0.6667

and decrease in CAM values (11%-36%) of classes in various consecutive versions of the investigated application packages. Thus, hypothesis H6 is accepted, while H5 and H7 are rejected.

6.2.1 Observations Corresponding to Cohesion Metrics

- The investigation of mean and median values of the three-cohesion metrics indicate poor cohesiveness of the CC when cohesion was evaluated using the LCOM metric. However, after analyzing LCOM3 metric and the CAM metric values, we observe better use of various variables in CC (low LCOM3 values) but poor use of parameter types (low CAM values) when compared to UCC.
- The analysis of median, minimum and maximum values indicate an LCOM3 value of greater than one, which represents poor class design with respect to cohesiveness. Certain classes also exhibited an extremely poor design with an LCOM3 value of 2 indicating compulsive restructuring of the class due to negligence of its cohesiveness.
- Majority of classes which evolved during the investigated versions of the Android software did not exhibit any change in the values of their cohesion metrics. However, the other trends indicate changes made to a class during evolution led to decrease in LCOM3 and CAM metric values and increase in LCOM values in the Android software. This means the classes exhibit better use of different variables but poor use of parameter types.

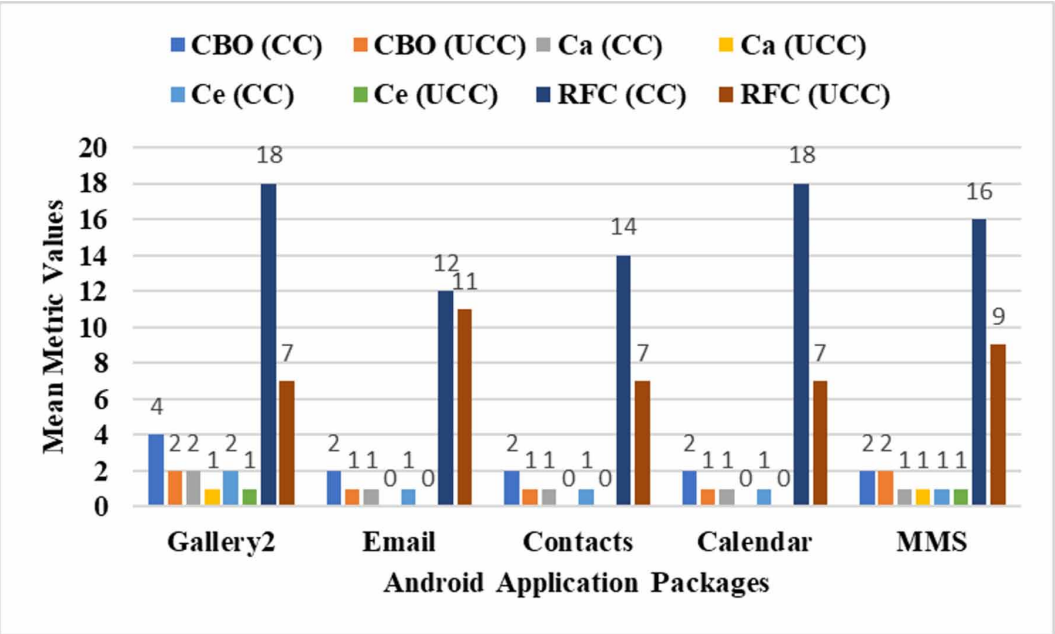
6.3 Trends of Coupling Metrics

The coupling dimension was categorized by four metrics viz. CBO, Ca, Ce and RFC. Figure 3 represents the mean values of coupling metrics over all the investigated versions (4.0.2-4.3.1) of the Android software. We also analyzed the median values (Table 6), the minimum and the maximum values of the coupling metrics obtained by classes over all the five versions for both CC and UCC. After analyzing Figure 3, it was observed that the mean values of coupling metrics are higher for CC as compared to UCC in most of the cases. For instance, the mean CBO values of CC in Gallery2, Email, Contacts and Calendar packages was double the mean CBO values of UCC in these application

Table 5. Version specific cohesion metric trends

Cohesion Metrics	4.0.2-4.0.4			4.0.4-4.1.2			4.1.2-4.2.2			4.2.2-4.3.1		
	Const.	Inc.	Dec.	Const.	Inc.	Dec.	Const.	Inc.	Dec.	Const.	Inc.	Dec.
LCOM	51	35	4	266	112	53	417	66	13	70	42	15
LCOM3	52	4	34	272	53	106	422	13	61	72	14	41
CAM	49	12	29	246	74	111	415	27	54	65	28	34

Figure 3. Mean values of coupling metrics



packages. Similarly, the median values of CC were also greater than UCC in all the application packages. A similar trend was shown by the mean values of Ca, Ce and RFC coupling metrics. This indicates that a class with higher coupling values is prone to change in future versions. However, it may be noted that median Ca values were same i.e. 0 for both CC and UCC classes in all the application packages. This trend indicates that there were very few classes with a Ca value, indicating a low number of classes exhibiting export coupling in the application packages.

The CBO metric depicts a cumulative of Ca and Ce metrics in majority of the cases. In order to analyse the number of classes which exhibited export or import coupling, we observed the number of classes for each application package which attained a non-zero value for Ca or Ce metrics in the CC category. We found a larger number of classes exhibiting a non-zero Ce value than those exhibiting a non-zero Ca value. This indicates that while evolution, there were more number of classes exhibiting import coupling as compared to export coupling.

We also assessed classes which exhibited higher Ce values to assess whether they also exhibit high Ca values and vice versa. We also found that in majority of the cases, classes with highest Ca

Table 6. Median values of coupling metrics

Application Package	CBO (CC)	CBO (UCC)	Ca (CC)	Ca (UCC)	Ce (CC)	Ce (UCC)	RFC (CC)	RFC (UCC)
Gallery2	3	2	0	0	1	1	14	7
Email	1	0	0	0	1	0	8	7
Contacts	1	0	0	0	1	0	11	6
Calendar	1	0	0	0	1	0	13	6
MMS	2	1	0	0	1	0	9	5

values have quite low Ce values and vice versa. Only in very few cases, classes with both high Ca and Ce values were found.

Table 7 states the trends of coupling metrics. According to the table, the majority (76-98%) of Ca, Ce and CBO metrics exhibited a "constant" trend. However, for the RFC metric apart from the "constant" trend, the "increasing" trend was quite popular (26-39%) in all the version specific trends except for versions 4.1.2-4.2.2. It may be noted that change in class may not always lead to change in RFC values. According to the discussed trends, hypothesis H8, H9, H10, and H11 are rejected.

Table 7. Version specific coupling metric trends

Coupling Metrics	4.0.2-4.0.4			4.0.4-4.1.2			4.1.2-4.2.2			4.2.2-4.3.1		
	Const.	Inc.	Dec.	Const.	Inc.	Dec.	Const.	Inc.	Dec.	Const.	Inc.	Dec.
Ca	88	2	0	411	16	4	479	11	6	113	5	9
Ce	82	8	0	379	35	17	471	20	5	104	11	12
CBO	80	10	0	371	42	18	462	27	7	96	15	16
RFC	51	35	4	266	112	53	417	66	13	70	42	15

A Wilcoxon signed rank test was conducted at $\alpha = 0.05$ to test whether all mean coupling metric values i.e. CBO Ca, Ce and RFC (of all the five application packages) of CC and UCC are different. The computed p-value of the test was less than 0.001 signifying significant differences between mean coupling metric values of CC and UCC. The Wilcoxon test results symbolized higher coupling metric values for CC indicating evolution leads to tightly coupled classes.

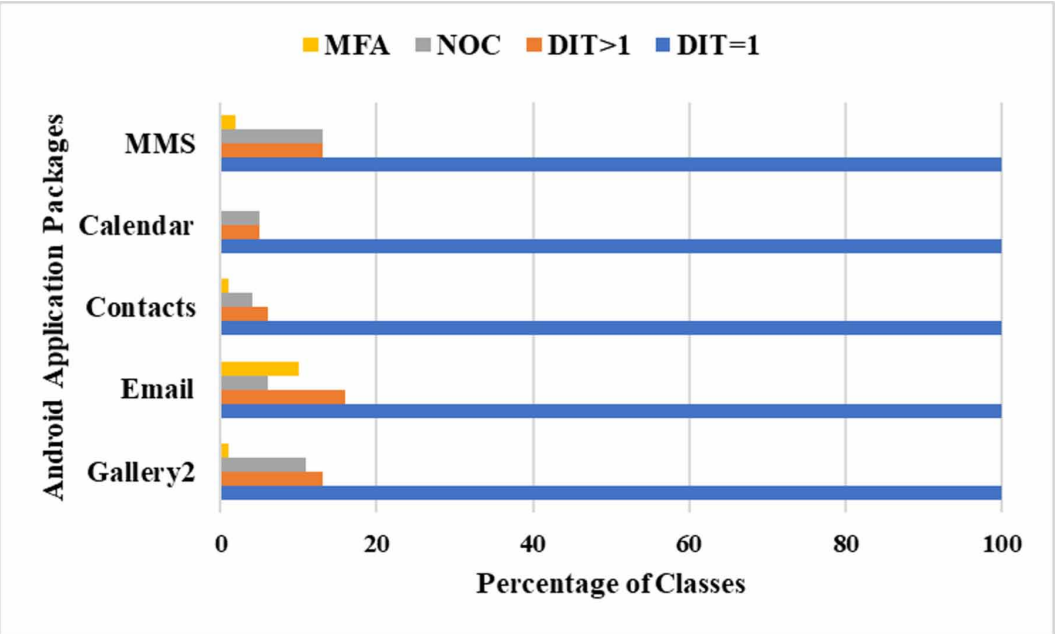
6.3.1 Observations Corresponding to Coupling Metrics

- The analysis of mean and median of coupling metrics indicates that a class with higher coupling values is prone to change in future versions as CC exhibited higher coupling metric values.
- As number of classes with a non-zero value for Ce metric are higher than a non-zero value for Ca metric, it indicates that a higher number of classes have a tendency of being dependent on other classes (import coupling). However, this could be an issue as it leads to decreased maintainability, understandability and reusability.
- In general, the classes exhibited either high import coupling (a high Ce value) or a high export coupling characteristic (a high Ca value). It was rare to find a class with high values of both Ca and Ce. Such classes can be termed as critical classes as changes in them should be carefully administered as multiple classes can lead to a change in such classes. Moreover, a change in such a critical class would be propagated to a large number of other classes due to high outgoing dependency of such a class.
- Majority of classes did not exhibit any change in the values of their coupling metrics during evolution. However, if there was a change in a class, there are higher chances of increasing the dependency of classes during evolution rather than decreasing their dependencies on other classes.

6.4 Trends of Inheritance Metrics

The inheritance attribute of classes was characterized by three metrics DIT, NOC and MFA. Figure 4 depicts the percentage of classes which used any of the inheritance attribute (i.e. have a non-zero value) in any of the five investigated versions. According to the figure, all the classes used the DIT attribute in corresponding application packages as all classes are descendant of the object class (Java

Figure 4. Classes exhibiting inheritance attributes



language). Therefore, we analysed percentage of classes with DIT value greater than 1, to understand its actual usage. Figure 4 shows that 5%-16% of classes exhibited a DIT value of greater than 1. The NOC and MFA metrics were rarely used by classes (NOC:4%-13% & MFA: 0%-10%).

We also observed the minimum and maximum values obtained by the DIT, NOC and MFA metrics in all the datasets. It was found that the maximum values of inheritance metrics are higher for CC as compared to UCC. The mean and median values of inheritance metrics did not give much information as in majority of the cases the mean and median values of NOC was 0 and that of DIT was 1.

We analysed the number of classes which depicted a non-zero NOC value and a DIT value >1 for each application package for CC and UCC. The CC exhibited greater NOC and DIT values than UCC. Also, most of the classes were found having either one or two children, there were very few cases with classes having three or more children. Similarly, though there were certain number of classes having DIT value of 1 or 2, there were very few classes with DIT value of three or more. Only 1% of CC classes exhibited a non-zero value for MFA metrics.

According to the trends observed (Table 8) by actual changed classes in consecutive versions, majority (94-100%) of them did not exhibit any change in inheritance metric values and were "constant". Only very few classes (0-0.04%) depicted an "increasing" or "decreasing". Thus, we reject hypothesis H12, H13 and H14.

6.4.1 Observation Corresponding to Inheritance Metrics

- As the investigated application packages were developed in Java language, all the classes exhibit a DIT value of greater than one as all classes are derived from the Java object class. However, the number of classes exhibiting a DIT value of greater than one, a non-zero NOC value and a non-zero MFA value was very low. This indicates that inheritance is rarely used in the investigated datasets.
- An observation of DIT metric values indicates that there were few classes with DIT values of 2 and 3 and hardly any classes with DIT value of 4 and 5. This indicates that inheritance levels

Table 8. Version specific inheritance metric trends

Inheritance Metrics	4.0.2-4.0.4			4.0.4-4.1.2			4.1.2-4.2.2			4.2.2-4.3.1		
	Const.	Inc.	Dec.	Const.	Inc.	Dec.	Const.	Inc.	Dec.	Const.	Inc.	Dec.
DIT	90	0	0	430	0	1	495	0	1	124	3	0
NOC	90	0	0	423	6	2	493	3	0	120	2	5
MFA	90	0	0	430	0	1	496	0	0	126	1	0

don't go too deep in the investigated datasets. Similarly, an observation of NOC values indicate few classes with three or more children. Thus, classes in the investigated datasets rarely grow to large breadths. Generally, they only have one or two children.

- An analysis of maximum and minimum values of inheritance metrics indicate higher values exhibited by CC. This implies classes exhibiting higher inheritance characteristics are more prone to changes during evolution of the software.
- The trends observed by inheritance metrics indicate that 94%-100% of classes exhibited no change in their DIT and NOC values. Thus, it is rare that a change due to evolution might affect inheritance attribute of a class.

7. COMPARISON OF RESULTS WITH LITERATURE STUDIES

This section compares the results obtained in the study with the results obtained in existing literature studies. Table 9 reports the observation obtained from a specific literature study with respect to a software dimension and our corresponding results.

Table 9. Comparison with literature studies

Literature Study	Dimension	Observation from Literature Study	Our Results
Lee et al. (2007)	Size	Number of classes i.e. the overall size of the software increases gradually with each release.	We only evaluate the trends of common classes, the common CC tend to increase in size over various releases of the software, rather than decrease.
Singh and Bhattacherjee (2014)	Size	Increase in WMC values as the software evolves.	Classes tend to increase in size in terms of total methods (WMC).
Alenezi and Almustafa (2015)	Size	Increase in total LOC values of a software as it evolves.	Only trends in common classes were assessed. CC tend to increase in size in terms of LOC.
Alenezi and Zarour (2015, 2016)	Cohesion	Increase in LCOM, LCOM3 and CAM metric values during evolution.	Higher number of classes with decreasing LCOM3 and CAM values. The increasing trend was popular for only the LCOM metric values.
Alenezi and Zarour (2015, 2016)	Cohesion	Modularity measures (also with respect to cohesion) are not improving with time.	Certain classes were found with alarming LCOM3 values.
Mubarak et al. (2010)	Coupling	Presence of "key" classes which exhibit high fan-in (Ca) and fan-out (Ce).	Confirm the existence of "key" classes. We term it as "critical" classes. Such classes are very few in number and need to be administered carefully during evolution.
Nasseri et al. (2008) and Kaur (2015)	Inheritance	Higher probability that new classes in an evolving software are added at levels 1 or 2 in the inheritance hierarchy.	Most of the classes exhibited either a DIT value of 1 or 2 indicating shallow inheritance levels.

8. THREATS TO VALIDITY

In order to ensure the construct validity, we need to be certain that the selected metrics are accurate representations of their corresponding concepts (Zhou et al., 2009). Previous research studies (Briand et al., 1998; Briand et al., 1999) have investigated the accuracy of some of the metrics used in this study, ensuring construct validity. It may be the case that apart from the effect of the evolution on software structure, the metric trends may be influenced by developer experience or software domain. We have not accounted for the confounding effect of these factors. This may be a possible threat to internal validity. The study also uses Wilcoxon signed rank test, a non-parametric test to statistically validate the trends of size and coupling metrics. This reinforces the conclusion validity.

The results of the study are extracted by validating the various versions of Android software, which is a popular open-source operating system. Although, the results may hold valid for a variety of open-source systems, especially the ones belonging to the operating system domain, there is a need to further validate these metric trends on open-source systems belonging to other domains. This is essential to further enhance the external validity of the obtained results. As the study investigates open-source systems, it aids its replicability.

9. CONCLUSION

This study analyzed the evolution patterns of 14 OO metrics on five application packages of the Android software. The OO metrics were categorized into four dimensions corresponding to size, coupling, cohesion and inheritance. The metrics for common classes in five versions (4.0.2, 4.0.4, 4.1.2, 4.2.2, and 4.3.1) of each application package were evaluated and classes in each application package were categorized as Changed Classes (CC) or Unchanged Classes (UCC). The trends of actual changed classes between consecutive versions were categorized as "Constant", "Increasing" or "Decreasing". The observations can be summarized as follows:

- The CC exhibited higher chances of increase in class size, higher coupling values, better variable usage, poor use of parameter types and higher inheritance characteristics as compared to UCC.
- The most common trend of actual changed classes between consecutive versions was increase in size metrics (WMC, NPM and LOC), RFC coupling metric and LCOM cohesion metric, no change in inheritance and other coupling metrics (Ca, Ce, CBO), and decrease in LCOM3 and CAM cohesion metrics.
- It was observed that increase in class size was generally attributed to addition of new methods in a class. There is a higher probability that an import coupling would lead to increase in coupling characteristic of a class rather than an export coupling.
- Poor design of classes with respect to cohesion is attributed to inadequate use of parameter types. However, CC exhibited effective usage of class variables.
- It was observed that very few classes exhibited the inheritance attribute and classes which used inheritance, did not exhibit deep inheritance levels or large number of subclasses.

Thus, a class with lower coupling values and higher cohesion values is better so that changes during software evolution do not lead to excessive deterioration of the class structure. Software practitioners should also ensure that the size and complexity of a class is manageable so that changes due to software evolution can be easily incorporated. In future, we would like to explore how change in metric values affect various software attributes such as maintainability, understandability or fault density. Furthermore, the observed trends may be evaluated on other open-source software to enhance the generalizability of the obtained results.

REFERENCES

Alenezi, M., & Almustafa, K. (2015). Empirical analysis of the complexity evolution in open-source software systems. *International Journal of Hybrid Information Technology*, *8*(2), 257–266. doi:10.14257/ijhit.2015.8.2.24

Alenezi, M., & Zarour, M. (2015). Modularity measurement and evolution in object-oriented open-source projects. In *Proceedings of the International Conference on Engineering & MIS* (p. 16). ACM. doi:10.1145/2832987.2833013

Alenezi, M., & Zarour, M. (2016). Does software structures quality improve over software evolution? Evidences from open-source projects. *International Journal of Computer Science and Information Security*, *14*, 61–75.

Briand, L. C., Daly, J. W., & Wüst, J. (1998). A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, *3*(1), 65–117. doi:10.1023/A:1009783721306

Briand, L. C., Daly, J. W., & Wust, J. K. (1999). A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, *25*(1), 91–121. doi:10.1109/32.748920

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493. doi:10.1109/32.295895

Counsell, S., Swift, S., & Crampton, J. (2006). The interpretation and utility of three cohesion metrics for object-oriented design. *ACM Transactions on Software Engineering and Methodology*, *15*(2), 123–149. doi:10.1145/1131421.1131422

Elish, M. O., & Al-Rahman Al-Khiaty, M. (2013). A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software. *Journal of Software: Evolution and Process*, *25*(5), 407–437.

Fioravanti, F., & Nesi, P. (2001). Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems. *IEEE Transactions on Software Engineering*, *27*(12), 1062–1084. doi:10.1109/32.988708

Gupta, B. S. (1997). *A critique of cohesion measures in the object-oriented paradigm* [Master's thesis]. Michigan Technological University.

Henderson-Sellers, B. (1995). *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc.

Kaur, K. (2015, September). Class inheritance structures and software maintenance activities-an empirical analysis. In *Proceedings of the 2015 1st International Conference on Next Generation Computing Technologies (NGCT)* (pp. 681-685). IEEE. doi:10.1109/NGCT.2015.7375208

Lee, Y., Yang, J., & Chang, K. H. (2007). Metrics and evolution in open source software. In *Proceedings of the Seventh International Conference on Quality Software QSIC'07* (pp. 191-197). IEEE. doi:10.1109/QSIC.2007.4385495

Lu, H., Zhou, Y., Xu, B., Leung, H., & Chen, L. (2012). The ability of object-oriented metrics to predict change-proneness: A meta-analysis. *Empirical Software Engineering*, *17*(3), 200–242. doi:10.1007/s10664-011-9170-z

Malhotra, R., & Khanna, M. (2017). An exploratory study for software change prediction in object-oriented systems using hybridized techniques. *Automated Software Engineering*, *24*(3), 673–717. doi:10.1007/s10515-016-0203-0

Malhotra, R., Pritam, N., Nagpal, K., & Upmanyu, P. (2014). Defect collection and reporting system for git based open source software. In *Proceedings of the International Conference on Data Mining and Intelligent Computing (ICDMIC)* (pp. 1-7). IEEE. doi:10.1109/ICDMIC.2014.6954234

Martin, R. C. (2002). *Agile software development: principles, patterns, and practices*. Prentice Hall.

Metrics Tool. (n.d.). Retrieved from http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/metric.html

Mubarak, A., Counsell, S., & Hierons, R. M. (2010). An evolutionary study of fan-in and fan-out metrics in OSS. In *Proceedings of the Fourth International Conference on Research Challenges in Information Science (RCIS)* (pp. 473-482). IEEE. doi:10.1109/RCIS.2010.5507329

Nasseri, E., Counsell, S., & Shepperd, M. (2008). An empirical study of evolution of inheritance in Java OSS. In *Proceedings of the 19th Australian Conference on Software Engineering ASWEC* (pp. 269-278). IEEE. doi:10.1109/ASWEC.2008.4483215

Saini, V., Sajnani, H., & Lopes, C. (2018). Cloned and non-cloned Java methods: A comparative study. *Empirical Software Engineering*, *23*(4), 2232–2278. doi:10.1007/s10664-017-9572-7

Singh, V., & Bhattacherjee, V. (2014). A new measure of code complexity during software evolution: A case study. *International Journal of Multimedia & Ubiquitous Engineering*, *9*(7), 257–266. doi:10.14257/ijmue.2014.9.7.34

Thamburaj, T. F., & Aloysius, A. (2017). Models for Maintenance Effort Prediction with Object-Oriented Cognitive Complexity Metrics. In *Proceedings of the World Congress on Computing and Communication Technologies (WCCCT)* (pp. 191-194). IEEE. doi:10.1109/WCCCT.2016.54

Zhou, Y., Leung, H., & Xu, B. (2009). Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness. *IEEE Transactions on Software Engineering*, *35*(5), 607–623. doi:10.1109/TSE.2009.32

Ruchika Malhotra is an Associate Professor in the discipline of Software Engineering, Department of Computer Science & Engineering, Delhi Technological University (formerly Delhi College of Engineering), Delhi, India. She is an Associate Dean in Industrial Research and Development, Delhi Technological University. She has been awarded prestigious UGC Raman Postdoctoral Fellowship by the Indian government for pursuing postdoctoral research from the Department of Computer and Information Science, Indiana University-Purdue University Indianapolis (2014-15), Indianapolis, Indiana, USA. She received her master's and doctorate degree in software engineering from the University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. She was an Assistant Professor at the University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. She has received IBM Faculty Award 2013. She is the recipient of the Commendable Research Award by the Delhi Technological University for her research in the year 2017 and 2018. She is the author of book titled "Empirical Research in Software Engineering" published by CRC press and co-author of a book on Object Oriented Software Engineering published by PHI Learning. Her research interests are in software testing, improving software quality, statistical and adaptive prediction models, software metrics and the definition and validation of software metrics. Her h-index is 27 as reported by Google Scholar. She has published more than 165 research papers in international journals and conferences.

Megha Khanna is currently working as an Assistant Professor in Sri Guru Gobind Singh College of Commerce, University of Delhi. She completed her doctoral degree from Delhi Technological University in 2019 and her master's degree in software engineering in 2010 from the University School of Information Technology, Guru Gobind Singh Indraprastha University, India. She received her graduation degree in Computer Science (Hons.) in 2007 from Acharya Narendra Dev College, University of Delhi. She is the recipient of Commendable Research Award by Delhi Technological University for her research in the year 2017 and 2018. She was also awarded the "Research Incentive" for her research in the year 2018 by the Governing body of Sri Guru Gobind Singh College of Commerce. Her research interests are in software quality improvement, applications of machine learning techniques in change prediction, and the definition and validation of software metrics. She has various publications in international conferences and journals.