An Empirical Study on the Impact of Refactoring on Quality Metrics in Android Applications

Oumayma Hamdi*, Ali Ouni*, Eman Abdullah AlOmar†, Mel Ó Cinnéide‡, Mohamed Wiem Mkaouer†
*ETS Montreal, University of Quebec, QC, Canada
†Rochester Institute of Technology, Rochester, NY, USA
‡School of Computer Science, University College Dublin, Ireland
omayma.hamdi.1@etsmtl.ca, ali.ouni@etsmtl.ca, eaa6167@g.rit.edu, mel.ocinneide@ucd.ie, mwmvse@rit.edu

Abstract—**Mobile applications must continuously evolve, sometimes under such time pressure that poor design or implementation choices are made, which inevitably result in structural software quality problems. Refactoring is the widely-accepted approach to ameliorating such quality problems. While the impact of refactoring on software quality has been widely studied in object-oriented software, its impact is still unclear in the context of mobile apps. This paper reports on the first empirical study that aims to address this gap. We conduct a large empirical study that analyses the evolution history of 300 open-source Android apps exhibiting a total of 42,181 refactoring operations. We analyze the impact of these refactoring operations on 10 common quality metrics using a causal inference method based on the Difference-in-Differences (DiD) model. Our results indicate that when refactoring affects the metrics it generally improves them. In many cases refactoring has no significant impact on the metrics, whereas one metric (LCOM) deteriorates overall as a result of refactoring. These findings provide practical insights into the current practice of refactoring in the context of Android app development.**

Index Terms—**Mobile app, refactoring, quality metrics, Android, empirical study**

I. Introduction

Android applications undergo modifications, improvements and enhancements to cope with rapid and evolving user requirements. Such maintenance activities can cause quality to decrease if improperly conducted [24], [35], [42], [53]. In order to facilitate software evolution, developers need to improve software structure on a regular basis. Refactoring is the most common approach to improve the internal structure of software systems without affecting their external behavior [6], [21], [34], [37], [40].

Mobile apps differ significantly from traditional software systems [27], [31], [33] in having to deal with limitations on specific hardware resources like memory, CPU, display size, etc., as well as the highly dynamic nature of the mobile app market and the ever-increasing user requirements. These differences can play an important role in mobile app development and evolution. Indeed, unlike object-oriented software systems [6], [8], [13], [15], [38], [48], the impact of refactoring on quality metrics in mobile apps has received little attention. Hence, much uncertainty exists about the relationship between refactoring and quality aspects in mobile apps. Yet, refactoring practices may exhibit different challenges in the context of Android apps. Even though refactoring aims at improving code structure, this expectation might not be always met in real settings as refactoring changes are often performed quickly to meet users requirements, fix defects or adapt to environment changes in the highly volatile mobile market [15]. To develop efficient and reliable refactoring support tools for mobile apps, there is a need to better understand the current refactoring practice and its impact on structural quality.

To fill this gap and improve the current knowledge about the impact of refactoring on structural quality, we conduct an empirical study on a dataset composed of 300 open-source Android apps that are freely distributed in the Google Play Store. We analyze the impact of 10 commonly used refactoring operations on 10 well-known quality metrics in Android apps. We identified a total of 42,181 applied refactoring operations and measured quality metrics values before and after each refactoring operation. Then we analyzed the impact of each refactoring on the considered quality metrics using a causal inference method based on the Difference-in-Differences (DiD) model, one of the widely-used analytical techniques for causal inference [9].

Overall, our study findings can be summarized as follows.

- Some refactoring types correlate with a broad improvement in software metric values. For example, the Move Method refactoring brings about a significant improvement in terms of coupling (CBO and RFC), cohesion (LCOM, TCC and LCC), complexity (WMC) and design size (LOC).
- The cohesion metric LCOM proved to be least consistent metric, improving for some refactoring types while deteriorating for others, and exhibiting an overall deterioration in response to refactoring. This resonates with earlier work showing LCOM to be very volatile under refactoring [38], and inclined to deteriorate when coupling improves [41].
- Non-refactoring code changes tend to have a negligible impact on the majority of quality metrics, except for the design size-related metrics which tend to increase in most of the commits. It is to be expected that design size related metrics increase over time as a project evolves, following Lehman's law on software evolution [28].

The paper is structured as follows. Section II reviews related work. Section III presents the design of our empirical study, while Section IV presents and discusses our results. Section V discusses the implications of our findings for developers, researchers and tool builders. Section VI describes the threats to validity of our study. Finally, Section VII draws our conclusions and present some ideas for future work.

II. Related work

Several research efforts have focused on studying when and how to apply refactoring. Fowler defined the first refactoring catalog that contains 72 refactoring operations and specified a guide containing information on when and how to apply them [21]. Later, Simon et al. [46] presented a generic approach to generate visualizations that supports developers to identify bad smells and propose adequate refactorings. They focus on use relations to propose move method/attribute and extract/inline class refactorings. They define a distance-based cohesion metric, which measures the cohesion between attributes and methods with the aim of identifying methods that use or are used by more features of another class than the class that they belong to, and attributes that are used by more methods of another class than the class that they belong to. The calculated distances are visualized in a three-dimensional perspective supporting the developer to manually identify refactoring opportunities.

Various research works attempted to quantitatively evaluate whether refactoring indeed improves quality in traditional software systems. Alshayeb et al. [8] investigated the impact of refactoring operations on five quality attributes, namely adaptability, maintainability, understandability, reusability, and testability. Their results highlight that benefits brought by refactoring operations on some code classes are often counterbalanced by a decrease of quality in some other classes. Pantiuchina et al. [43] explored the correlation between code metrics and the quality improvement explicitly reported by developers in commit messages. The study shows that quality metrics sometimes do not capture the quality improvement documented by developers. Similarly, AlOmar et al. [6] conducted a large scale empirical study on open-source java projects to investigate the extent to which refactorings impact on quality metrics match with the developers perception. The study results indicate that quality metrics related to cohesion, coupling and complexity capture more developer intentions of quality improvement than metrics related to encapsulation, abstraction, polymorphism and design size

Tahvildari & Kontogiannis [50] analyzed the association of refactorings with a possible effect on maintainability enhancements through refactorings. They use a catalogue of object-oriented metrics as an indicator for the transformations to be applied to improve the quality of a legacy system. The indicator is achieved by analysing the impact of each refactoring on these object-oriented metrics. Ó Cinnéide et al. [38] investigated the impact of refactoring on five popular cohesion metrics using eight Java desktop systems. Their results demonstrate that cohesion metrics disagree with each other in 55% of cases. Furthermore, Geppert et al. [22] empirically investigated the impact of refactoring on changeability. They considered three factors for changeability: customer reported defect rates, effort, and scope of changes. Szoke et al. [49] performed a study on five software systems to investigate the relationship between refactoring and code quality. They show that small refactoring operations performed in isolation rarely impact software quality. On the other side, a high number of refactoring operations performed in block helps in substantially improving code quality.

Strogglos and Spinellis [48] investigated the impact of refactoring on eight OO quality metrics. Their results indicate that refactoring caused a non-trivial increase in some specific metrics such as LCOM, Ca, RFC leading to less coherent classes or assigning more responsibilities to other classes. Kataoka et al. [26] studied the refactoring effect on various coupling metrics, comparing the metrics before and and after the refactorings Extract Method and Extract Class, which were performed by a single developer in desktop C++ programs. More recently, Cedrim et al. [15] conducted a longitudinal study of 25 desktop projects to examine the impact of refactoring on software quality. The results indicate that only 2.24% of refactorings removed code smells while 2.66% of the refactorings introduced new ones. For the sake of clarity, Table I provides a summary of the existing works.

We observe from the existing literature that most studies focus basically on desktop applications while little knowledge is available for mobile apps. Furthermore, existing studies are merely limited to some particular quality metrics, or/and few refactoring types. In our study, we focus on Android apps while considering the analysis of more quality metrics. While current studies collect refactorings based on mining developers documentation, or release-based static analysis tools, we use a fine grained detection of refactoring based on RefactoringMiner to reduce any bias towards imprecise collection of refactorings. Furthermore, one of the limitations in the state-of-the-art studies is that they do not consider that refactoring operations are typically accompanied with other code changes in either the commit [6], [15], [48] or release levels [8], [13], [16]. Such code changes can add more noise to the analyzed quality metrics values, and impact the final outcome the metrics analysis. In our study, we adopt a causal inference based on the DiD model [9] to better assess the impact of refactoring on quality metrics and assure that the metrics variations are due to refactoring.

III. Study Design

This section describes the design of our empirical study. We first setup our research question. Then, we explain our experimental setup including data collection, and analysis methods employed.

Our main goal in this study is to investigate how refactoring affects structural quality metrics in Android apps. In particular, we address the following research question:

RQ. *Do refactorings applied by Android developers improve quality metrics?*

TABLE I: A summary of the literature on the impact of refactoring on software quality attributes.

Study	Year	Software Metric	Internal Quality Attribute	Refactoring Level Class	Refactoring Level Method	Refactoring Level Field
Simon et al. [46]	2001	Cohesion measures	Cohesion	Yes	Yes	Yes
Kataoka et al. [26]	2002	Coupling measures	Coupling	Yes	Yes	No
Tahvildari & Kontogiannis [50]	2004	LCOM / WMC / RFC / NOM CDE / DAC / TCC	Inheritance / Cohesion / Coupling / Complexity	Not mentioned		
Geppert et al. [22]	2005	Not mentioned	Not mentioned	Not mentioned		
Stroggylos & Spinells [48]	2007	CK / Ca / NPM	Inheritance / Cohesion / Coupling / Complexity	Not mentioned		
Alshayeb [8]	2009	CK / LOC / FANOUT	Inheritance / Cohesion / Coupling / Size	Yes	Yes	Yes
Ó Cinnéide et al. [38]	2012	LSCC / TCC / CC / SCOM / LCOM5	Cohesion	Yes	Yes	Yes
Szoke et al. [49]	2014	CC / U / NOA / NII / NAni LOC / NUMPAR / NMni / NA	Size / Complexity	Not mentioned		
Cedrim at al. [15]	2016	LOC / CBO / NOM / CC FANOUT / FANIN	Cohesion / Coupling / Complexity	Yes	Yes	Yes
Pantiuchina et al. [43]	2018	LCOM / CBO / WMC / RFC C3 / B&W / SRead	Cohesion / Coupling / Complexity	Yes	Yes	Yes
AlOmar et al. [6]	2019	CK / FANIN / FANOUT / CC NIV / NIM Evg / NPath / MaxNest / IFANIN LOC / CLOC / CDL / STMTC	Inheritance / Cohesion / Coupling / Complexity Size / Polymorphism / Encapsulation / Abstraction	Yes	Yes	Yes

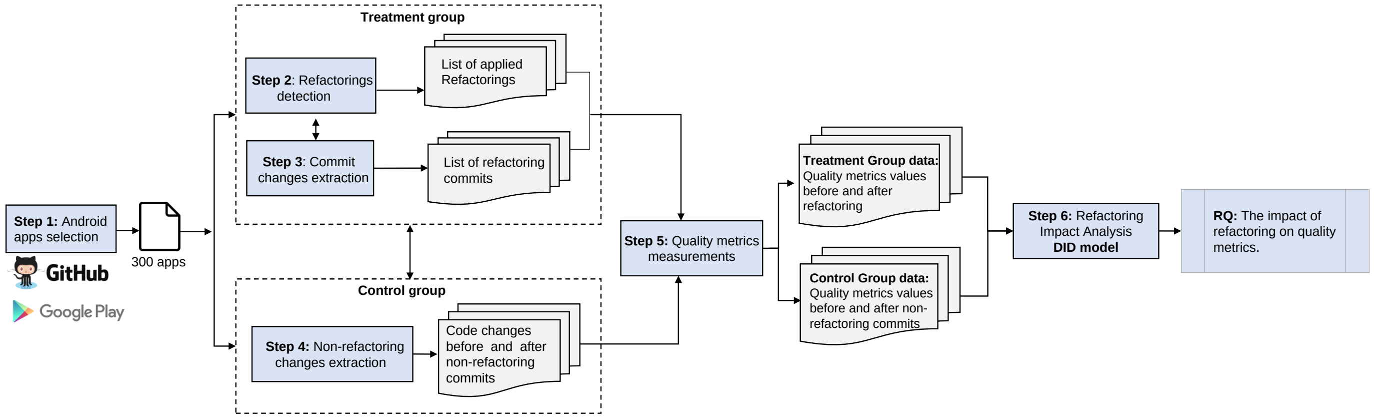

Treatment group
Step 2: Refactorings detection
List of applied Refactorings
Step 3: Commit changes extraction
List of refactoring commits
Step 1: Android apps selection
300 apps
GitHub
Google Play
Control group
Step 4: Non-refactoring changes extraction
Code changes before and after non-refactoring commits
Step 5: Quality metrics measurements
Treatment Group data: Quality metrics values before and after refactoring
Control Group data: Quality metrics values before and after non-refactoring commits
Step 6: Refactoring Impact Analysis DID model
RQ: The impact of refactoring on quality metrics.

Fig. 1: The overall process of our empirical study.

A. Context and Dataset

Since this study is focused on Android apps for which we investigate the relationship between refactoring occurring in the development cycle and quality metrics, the context of our study consists of a representative set of Android apps, and refactorings/quality metrics. In particular, we analyzed *(i)* 10 common refactoring types which are amongst Fowler's refactoring catalog [21], and *(ii)* a set of ten common quality metrics based on Chidamber & Kemerer (CK) metrics suite [17]. The CK metrics have been studied in several empirical studies and have shown their relevance in capturing different aspects of code maintainability [6], [12], [13], [23], [29], [38]. We also considered other common quality metrics such as number of lines of code (LOC), the Tight Class Cohesion (TCC), the Loose Class Cohesion (LCC), the number of static invocations (NOSI), and the variables quantity (VQTY) as they measure additional quality aspects from developers perspective. Tables II and III report the set of refactorings and quality metrics, respectively, that are investigated in our study.

B. Empirical Study Setup

To address our research question, we design a controlled experiment where we select two groups of code changes, a first group that consists of refactoring-related change changes (*i.e.*, treatment group), and a second that consists of a non-refactoring code changes (*i.e.*, control group). Thereafter, we investigate the impact of both groups on quality metrics to allow statistical analysis. Figure 1 describes the overall process of our study which consists of six main steps: (1) Android apps selection, (2) refactoring extraction, (3) commit extraction, (4) non-refactoring changes extraction, (5) Quality metrics measurement, and (6) refactoring impact analysis.

1) Step 1: Android apps selection: We target open-source Android apps that are freely distributed in the Google Play store and have their versioning history hosted on GitHub. For this purpose, we performed a custom search on GitHub by targeting all Java repositories in which the `readme.md` file contains a link to a Google Play Store page. Overall, we obtained 19,212 apps. Thereafter, inspired by previous works [19], [30], we applied the following filters to exclude:

- Apps whose GitHub repository does not contain an `AndroidManifest.xml` file as they clearly do not refer to real Android apps. The result of this filter was a collection of 5,766 apps.
- Apps for which the corresponding Google Play page is not existing anymore. This filter returned 3,160 apps.
- Repositories that contain forks of other repositories. This filtering step leads to a final set of 1,923 Android apps

TABLE II: The list of studied refactoring operations.

Ref.	Refactoring	Description	Level
MM	Move Method	Moves a method from a class to another class.	Method
EM	Extract Method	Creates a new method from an existing fragment of code from a given method.	Method
IM	Inline Method	Replaces calls to the method with the method's content and delete the method itself.	Method
EMM	Extract and Move Method	Extracts and moves method.	Method
PDM	Push Down Method	Moves a method from a class to those subclasses that require it.	Method
PUM	Pull Up Method	Moves a method from a class(es) to its immediate superclass.	Method
MA	Move Attribute	Moves Attribute from a class to another class.	Attribute
PDA	Push Down Attribute	Moves an attribute from a class to those subclasses that require it.	Attribute
PUA	Pull Up Attribute	Moves an attribute from a class(es) to their immediate superclass.	Attribute
ESC	Extract Super Class	Creates a superclass from two classes with common attributes and methods.	Class
MC	Move Class	Moves a class to another package.	Class

TABLE III: The list of studied quality attributes and metrics.

Quality Attribute	Metric	description
Coupling	Coupling Between Objects (CBO)	Number of classes that are coupled to a particular class [17].
	Number Of Static Invocations (NOSI)	Number of invocations of static methods [10]
	Response For a Class (RFC)	Number of method invocations in a class [17].
Cohesion	Lack Of Cohesion of Methods (LCOM)	Numbers of pairs of methods that shared references to instance variables [17].
	Tight Class Cohesion (TCC)	Numbers of directly connected public methods in a class [10].
	Loose Class Cohesion (LCC)	Numbers of directly/indirectly connected public methods in a class [10].
Complexity	Weight Method Class (WMC)	The sum of all the complexities of the methods (McCabe's cyclomatic complexity) in the class [17].
Design Size	Lines of code (LOC)	Number of lines of code ignoring spaces and comments [10].
	Variable Quantity (VQTY)	Number of declared variables [10].
Inheritance	Depth of Inheritance Tree (DIT)	Number of classes that a particular class inherits from [17].

Thereafter, we randomly selected a representative set of 300 apps which represents 15% of the final set, exhibiting a total of 42,181 refactoring operations. We focused our study to this set of apps for computational reasons. It is worth noting that the sample size of 300 apps and 42,181 refactoring operations is larger than related studies on the impact of refactoring on software quality [8], [13], [15], and than typical samples in software engineering research [55]. Table IV summarizes the statistics about the collected dataset.

2) Step 2: Refactorings detection: In this step, we collect all the refactoring operations applied to the studied apps. We utilize RefactoringMiner [52] to detect applied refactoring instances on the commit level. RefactoringMiner is a command-line based open source tool that is built on top of the UMLDiff [56] algorithm for differencing object-oriented models. RefactoringMiner has been shown to achieve a precision of 98% and recall of 87% [45], [52]. The tool walks through the commit history of a project's Git repository to extract refactorings between consecutive commits. RefactoringMiner supports the detection of various common refactoring types from Fowler's catalog. Among the supported refactorings, all refactoring types detected by Refactoring Miner were considered in this study, except the Rename Method and Rename Class refactorings as they are not directly related to one of the structural metrics studied in our study. Overall, our extraction process identifies a list of 10 common refactoring types which are amongst the most common refactoring types [6], [14], [15], [37], [40]. Tables II and V report the list and the number of refactorings, respectively, that are investigated in our study.

3) Step 3: Commit changes extraction: After the extraction of all refactoring operations, we collect the IDs of all refactoring commits, *i.e.*, commits in which a refactoring operation was applied, as well as the IDs of the commits that immediately precede the refactoring commit. The GitHub API facilitates this process; in particular, we use the `git clone` command to download the source code of each refactoring commit as well as its immediately preceding commit. These commits enable the identification of quality metrics values before and after the application of refactoring.

4) Step 4: Non-refactoring changes extraction: In this step, we extract a set of commits that contain non-refactoring changes for our controlled experiment. To do this, based on the treatment group, we randomly selected a set of non-refactoring commits representing our control group. For each commit, we collected its ID as well as the commits that precede it. Thereafter, we performed the same procedure adopted in Step 3 to collect their source code.

5) Step 5: Quality metrics measurement: To assess the impact of refactoring on software quality, we need to measure a set of quality metrics. In particular, we measure for each applied refactoring change as well as non-refactoring changes, the class level metrics before and after the change has been applied in the commit level. Specifically, since we already have the list of refactoring operations applied in each commit, we compute for each class the quality metric values before and after each commit in both treatment and control groups. To calculate the values of these metrics we utilized a widely-used open source CK Metrics Suite tool, namely, CK-metrics,which is a command-line based tool provided by Aniche [10] that allows automating our dataset collection process.

6) Step 6: Refactoring Impact Analysis: In this step, we investigate whether or not each metric is improved by refactoring. In order to do this, we set up two hypotheses, the null hypothesis H_0 assumes that a refactoring operation r_i does not improve a quality metric m_j, and the alternative hypothesis H indicates that the refactoring r_i improves m_j.

After collecting the metric values before and after each commit in both treatment and control groups, we calculate the differences between their quality metric values before and after the refactoring change, at the class level. Thereafter, we use two statistical methods (1) statistical significance, and (2) causal inference.

Statistical Significance Analysis. To capture the overall trends of the variation in the metric values we use statistical significance analysis. To do so, for each refactoring operation r_i, and each metrics m_j, we use the Wilcoxon rank-sum test [54], a non-parametric test, to assess the statistical differences between the distribution of m_j before and after the application of r_i. In addition to the Wilcoxon test, we used the non-parametric effect Cliff's delta (δ) [18] to compute the effect size, *i.e.*, the magnitude of the difference between the distributions. The value of effect size is statistically interpreted as:

- *Negligible* : if $|\ \delta\ | < 0.147$,
- *Small* : if $0.147 \leq |\ \delta\ | < 0.33$,
- *Medium* : if $0.33 \leq |\ \delta\ | < 0.474$, or
- *High* : if $|\ \delta\ | \geq 0.474$.

Furthermore, to better assess the impact of a specific refactoring operation on quality metrics, we performed a causal inference experiment to assess whether the metrics variations are due to the refactoring changes or to other code changes.

Causal Inference Analysis. Causal inference stems from the social sciences and explores cause and effects as its main concern [9]. In econometrics, difference-in-differences (DiD) methods are one of the key analytical elements for causal inference. DiD is used to statistically analyze actual and counterfactual scenarios, thereby enabling a causality analysis. To investigate the effects of a treatment in statistics, one cannot see the results with and without an intervention based on one individual only. As shown in Figure 2, the DiD model addresses this problem by comparing two groups, (1) a group with the intervention, called *treatment* group (*i.e.*, a set of code changes with refactoring) and (2) a group without it, called a *control* group (*i.e.*, a set of code changes without refactoring). The underlying assumption of DiD design is that the trend of the control group provides an adequate proxy for the trend that would have been observed in the treatment group in the absence of treatment. Let, T and C, the treatment and the control group, respectively. The refactoring impact *RI* of a given refactoring operation R on a given quality metric M_i is calculated as follows:

$$RI(R, M_j) = Y^R_{M_i} - Y^C_{M_i} \qquad (1)$$

where $Y^R_{M_i}$ is the median perceived impact after the application of the set of refactorings R on the treatment group T on the metric M_i; and $Y^C_{M_i}$ is the median perceived change in the control group C on the metric M_i.

C. Replication package

Our dataset is available in our replication package for future replications and extensions [44].

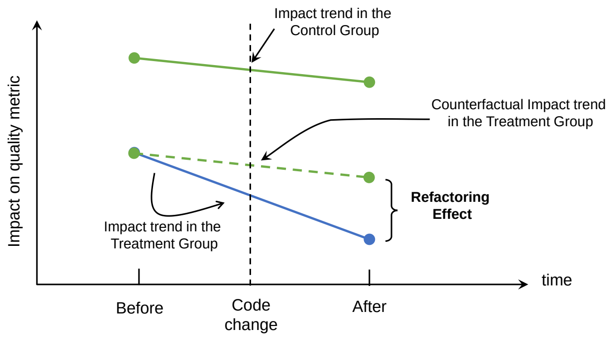


Fig. 2: An example of the causal inference method using a DiD model showing the refactoring impact on a quality metric before versus after the application of refactoring.

TABLE IV: Dataset statistics.

Statistic	**Count**
# of Android apps	300
# of commits with refactorings	13,500
# of refactoring operations	42,181
Total number of commits	271,263

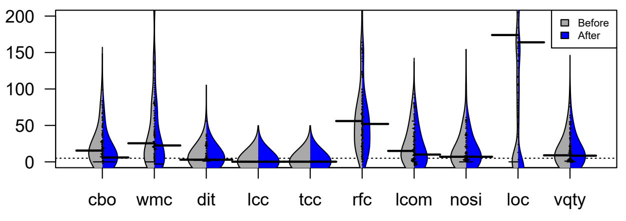
IV. Empirical Study Results

This section reports and discusses our experimental results to address our research question: *do refactorings applied by Android developers improve quality metrics?* To answer this question, for each commit change of both groups, described in III, we compute its corresponding metric values before and after each commit in both treatment and control groups. Figures 3 and 4 show the general distribution of the metrics values before and after commit changes in the treatment, and control groups, respectively. We also provide a detailed analysis in Table VI where each column reports *(i)* the impact of the respective refactoring type based on the DiD technique using Equation 1, *(ii)* the predominant behavior indicating whether the refactoring impact is positive or negative, and *(iii)* the p-value as well as the Cliff's delta (δ).

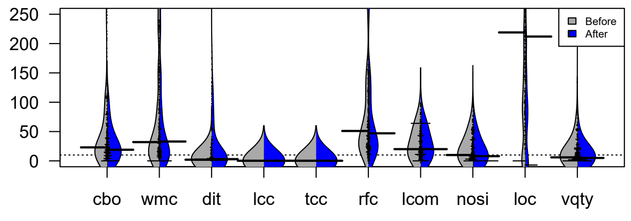
In the following, we report and discuss the obtained results for each quality metric along with real world examples from our experiments.

TABLE V: The list of refactoring applied to the analyzed apps.

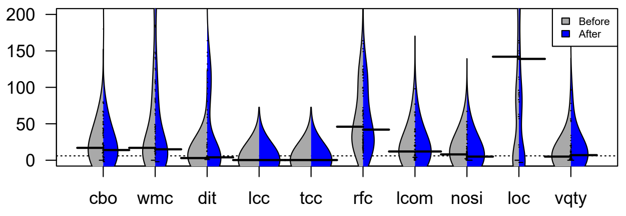
Refactoring type	Number
Extract Method	11,736
Move Attribute	8,321
Move Method	5,847
Extract And Move Method	5,121
Inline Method	3,952
Push Down Method	2,541
Pull Up Attribute	1,371
Pull Up Method	1,170
Extract SuperClass	1,140
Move Class	982
Total	42,181



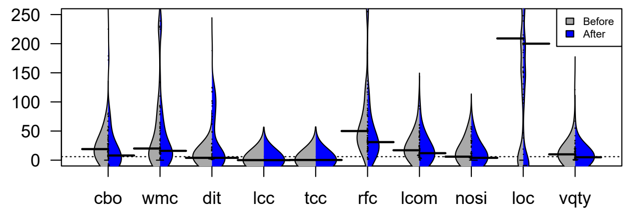
(a) Move Method



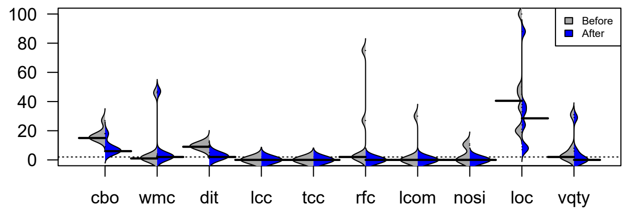
(b) Extract Method



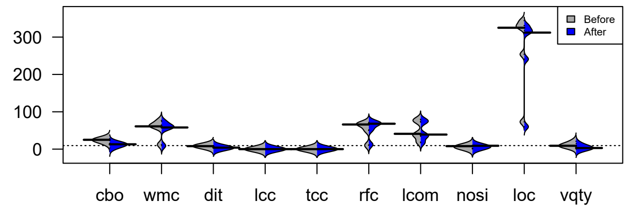
(c) Inline Method.



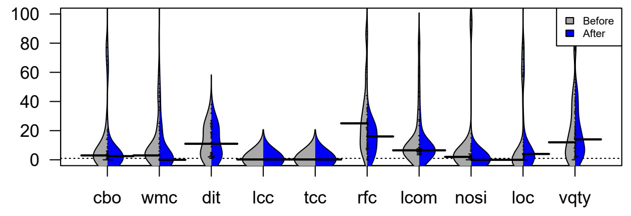
(d) Extract And Move Method



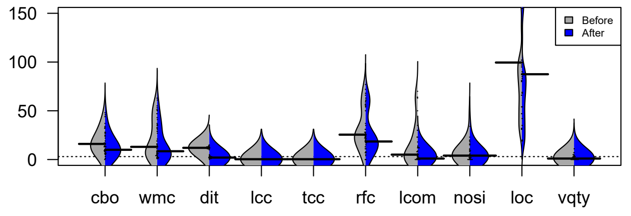
(e) Push Down Method.



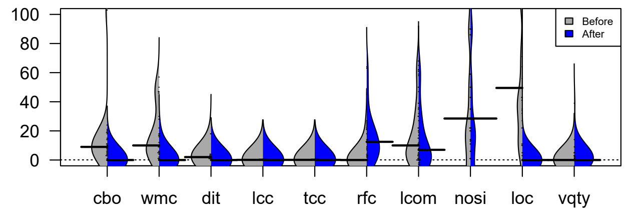
(f) Pull Up Method.



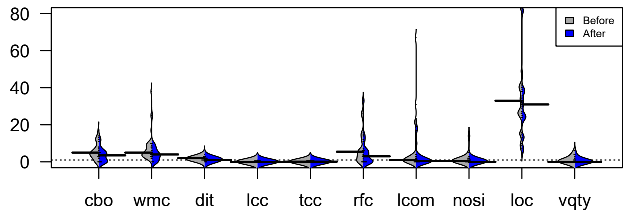
(g) Move Attribute.



(h) Pull Up Attribute.



(i) Extract Super Class.



(j) Move Class.

Fig. 3: Treatment group results: Beanplots of metric values before and after each refactoring operation.

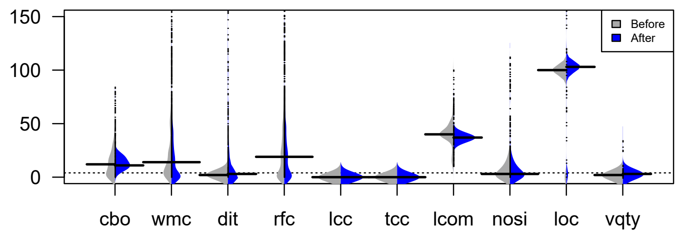


Fig. 4: Control group results: The impact of non-refactoring code changes on quality metrics.

A. *Results for Coupling Metrics*

Coupling is defined as the strength of the dependencies that exist between classes [17], [47]. Low coupling is desirable since it helps in isolating responsibilities and changes. As shown in Table III, we assess three coupling metrics. The first is the Coupling Between Object (CBO), counting the number of dependencies a class has (*i.e.*, the number of other classes it depends on). The second metric is the Response for a Class (RFC), calculated as the number of distinct methods and constructors invoked by a class. The third is Number Of Static Invocations (NOSI) which counts the number of invocations of static methods. The higher the CBO, RFC and NOSI the worse is the class coupling.

1) CBO: From the beanplots in Figure 3 and Table VI, we observe that several applied refactorings improve the CBO metric (*i.e.*, decrease its value). The most influential refactorings are *Pull Up Method*, *Push Down Method*, and *Extract Super Class* significantly reducing CBO from 9 to

TABLE VI: The impact of refactoring (treatment group) and non-refactoring (control group) changes on quality metrics.

Data	Change	Measure	Coupling			Cohesion			Complexity	Design Size		Inheritance
			CBO	RFC	NOSI	LCOM	TCC	LCC	WMC	LOC	VQTY	DIT
Treatment Group	Extract Method	Refactoring impact	0	-4	0	3	0	0	-2	-4	0	0
		Behavior	-	↓	-	↑	-	-	↓	↓	-	-
		P-value (δ)	0.06 (S)	<0.05 (S)	0.10 (S)	0.07 (N)	0.18 (S)	0.18 (S)	<0.05 (S)	<0.05 (S)	0.06 (N)	0.06 (N)
	Move Attribute	Refactoring impact	0	-9	5	1	0.1	0.1	0	0	-12	0
		Behavior	-	↓	↑	↑	↑	↑	-	-	↓	-
		P-value (δ)	<0.05 (N)	<0.05 (N)	0.17 (N)	<0.05 (N)	<0.05 (N)	<0.05(N)	0.40 (N)	0.13 (N)	<0.05 (S)	0.1 (N)
	Move Method	Refactoring impact	-8	-4	0	-2	0.3	0.3	-3	-13	0	0
		Behavior	↓	↓	-	↓	↑	↑	↓	↓	-	-
		P-value (δ)	<0.05 (S)	<0.05 (N)	0.70 (N)	<0.05 (S)	<0.05 (N)	<0.05 (N)	<0.05 (S)	<0.05 (M)	0.11 (N) (M)	0.15 (S)
	Extract And Move Method	Refactoring impact	-8	-11	0	-2	0.7	0.7	-3	-12	0	0
		Behavior	↓	↓	-	↓	↑	↑	↓	↓	-	-
		P-value (δ)	<0.05 (S)	<0.05 (N)	0.33 (N)	<0.05 (S)	<0.05 (N)	<0.05 (N)	<0.05 (S)	<0.05 (M)	0.08 (N)	0.07 (N)
	Inline Method	Refactoring impact	-3	0	0	-4	0	0	-1	0	-5	0
		Behavior	↓	-	-	↓	-	-	↓	-	↓	-
		P-value (δ)	<0.05 (N)	0.82 (N)	1 (N)	0.31 (N)	0.10 (N)	0.10 (N)	0.17 (S)	1(N)	0.59 (N)	1 (N)
	Push Down Method	Refactoring impact	-10	0	0	3	0	0	1	-15	-3	-7
		Behavior	↓	-	-	↑	-	-	↑	↓	↓	↓
		P-value (δ)	<0.05 (M)	1 (N)	1 (S)	1 (N)	1 (N)	1 (N)	0.10 (N)	<0.05 (M)	<0.05 (N)	<0.05 (S)
	Pull Up Attribute	Refactoring impact	-4	0	0	0	0	0	-2	-9	0	-10
		Behavior	↓	-	-	-	-	-	↓	↓	-	↓
		P-value (δ)	<0.05 (S)	1 (N)	1 (N)	0.08 (N)	<0.05 (N)	<0.05 (N)	<0.05(N)	<0.05 (S)	0.08 (N)	<0.05 (M)
	Pull Up Method	Refactoring impact	-11	2	1	2	0.01	0.01	-3	-16	-7	-5
		Behavior	↓	↑	↑	↑	↑	↑	↓	↓	↓	↓
		P-value (δ)	<0.05 (M)	0.51 (N)	1 (N)	<0.05 (N)	1 (N)	1 (N)	0.03 (N)	<0.05 (M)	<0.05 (S)	<0.05 (S)
	Extract Super Class	Refactoring impact	-9	-8	0	0	0	0	-10	-24	-1	-8
		Behavior	↓	↓	-	-	-	-	↓	↓	↓	↓
		P-value (δ)	<0.05(M)	<0.05 (S)	0.88 (N)	0.09 (N)	1 (N)	1 (N)	<0.05 (M)	<0.05 (M)	1 (N)	<0.05 (S)
	Move Class	Refactoring impact	0	0	-1	2	0	0	-1	-6	-1	-3
		Behavior	-	-	↓	↑	-	-	↓	↓	↓	↓
		P-value (δ)	<0.05 (S)	1 (N)	0.17 (N)	<0.05 (N)	1 (N)	1 (N)	<0.05 (N)	<0.05 (N)	0.10 (N)	<0.05 (S)
Control Group	Commit change	Change commit	-1	0	0	-3	0	0	0	10	1	0
		Behavior	↓	-	-	↓	-	-	-	↑	↑	-
		P-value (δ)	0.08 (N)	0.11 (N)	0.73 (N)	0.10 (N)	0.97 (N)	0.97 (N)	0.07 (N)	<0.05 (S)	<0.05 (N)	0.34 (N)

Legend:
Metric improvement: Low 🟦🟦🟦🟦🟦🟦🟦🟦 High
Metric disprovement: Low 🟥🟥🟥🟥🟥🟥🟥🟥 High

Effect size: L: Large, M: Medium, S: Small, N: Negligible
Behavior: "↑" : indicates that the metrics increased; "↓" : indicates that the metric decreased; "-" : indicates that the metric remains unaffected.

11 with a medium effect size. These refactorings are typically applied when classes or subclasses grow and develop independently of one another, causing identical (or similar) methods each having its own dependencies. They often help reducing duplicate code, replacing inheritance with delegation and vice versa as well as reducing dependencies through polymorphism. Furthermore, *Move Method* or *Extract And Move Method* tend to significantly improve CBO by a median value of 8, each with a small effect size. Typically, these method-level move refactorings help organizing functionalities across classes and thus reduce dependencies between them. Overall, as shown in Table VI, the CBO variation for all refactorings is significant and accompanied with a medium or small effect size depending on the refactoring type. Particularly, the effect size is negligible for the *Inline Method*, as it could be applied either in methods from the same class or from different classes. Only the latter can help reducing coupling.

2) RFC: As it can be seen from the beanplots in Figure 3 and Table III, *Extract And Move Method*, and *Move Attribute* are the most influential refactoring that improve RFC by 11 and 9, respectively. Moreover, *Extract Super Class* have shown to improve RFC by a median score of 8, while less impact is observed by both *Extract Method* and *Move Method* refactorings with a median of 4, each.

3) NOSI: From Figure 3 and Table VI, we observe that the NOSI metric has not been impacted by any of the applied refactorings since when comparing the distributions of values before and after refactoring, no statistically significant difference is observed. This is not very surprising, as most of refactorings do not have a direct impact on static methods.

It is worth noting that our findings in Android apps share some similarities with desktop apps for the coupling which is positively impacted after applying the refactoring [36], [43]. Moreover, the the *Extract Method* and *Move Method* refactorings are applied in both Android and desktop applications to improve the coupling [13], [20].

To observe the salient impacts of refactoring on coupling, we refer to a real world example from our dataset showing the impact of a *Move Method* refactoring on coupling metrics, from the WordPress-Android app, in the commit [1]. The commit's refactoring consists of moving of the method `showJetpackSettings()` from the class `EditPostActivity` to the class `ActivityLauncher`. Interestingly, this Move Method refactoring resulted in a coupling reduction for the `ActivityLauncher` class, with a drop of its CBO from 18 to 13 and its RFC from 33 to 31. Looking deeper into the the source code to understand the reason behind these improvements, we find that the "*Start Jetpack security settings*" activity was initially launched from the `ActivityLauncher` class via the `startActivity()` method. This method use the `intent` object to start this activity. However, `intent` was intitially implemented in the `showJetpackSettings()` method in the *EditPostActivity*. As consequence, each time this activity is launched, the class `ActivityLauncher` calls the `showJetpackSettings()` method. Thus, this refactoring helped moving the method to the class that uses it most which decreased the number of dependencies between both classes, resulting in an improvement in both CBO and RFC.

Finding 1. *Refactoring has a significant positive impact on coupling in terms of both the CBO and RFC metrics, while no significant impact was found on the NOSI metric. The most influential refactorings that promote low coupling are "Move Method", and "Extract And Move Method".*

B. Results for Cohesion Metrics

Cohesion assesses the degree to which the responsibilities implemented in a class belong together [47]. High cohesion is desirable since it promotes encapsulation and adherence to the Single Responsibility Principle, one of the SOLID design principles [25]. In this study, we consider three cohesion metrics, the first metric is the normalized [39], considering the shared instance variables between method pairs of a class. If the value of this metric is low, it indicates a strong cohesiveness of the class. The second cohesion metric is TCC, which considers the direct connection of public methods in a class. The third is LCC, which is similar to TCC but additionally considers the indirect connection of public methods in a class. TCC and LCC provide another way of measuring the cohesiveness of a class. The higher the TCC and LCC a values are, the more cohesive is the class. It is anticipated that cohesion may be improved by moving-related refactoring operations. In general, moving a method that does not access local attributes or methods, or is called by few local methods improves cohesion.

1) LCOM: The beanplots from Figure 3 and Table VI show that LCOM is improved when applying *Move Method* and *Extract And Move Method* refactorings. For both refactorings, the median values significantly decreased by 2, even though they are accompanied by a small effect size. However, we also observe from the results in Table VI that the *Move Attribute*, *Pull Up Method* and *Move Class* refactorings caused LCOM metric to disprove by a median of 1 and 2 with a negligible effect size. These results suggest that the LCOM metric can either improve or disprove after refactoring, and developers need to pay attention to cohesion when modifying their code and use appropriate refactoring operations.It is worth noting that our results match previous work observations for the cohesion, *i.e.*, cohesion worsens rather than improves after the refactoring application [20], [48].

2) TCC: As can be seen in beanplots of Figure 3 and Table VI, *Extract And Move Method* is the most influential refactoring on TCC which improve it by 0.7. Whereas, *Move Method* and *Move Attribute* refactorings tend to have less impact on the metrics with a median improvement of 0.3, 0.1, respectively. It is worth noting that the differences are statistically significant even though with negligible effect size.

3) LCC: We found that LCC achieves similar results to the TCC metric. This was expected since both metrics reflect similar cohesion characteristic as mentioned earlier in Section IV-B, except that LCC further involves the number of indirect connections between visible classes. Thus, the constraint $LCC \geq TCC$ holds always. Upon a qualitative investigation of our dataset, we observe that moving methods from one class to another is a popular and effective refactoring to improve cohesion, as it often involves adding a parameter when resources of the original class are used, and removing that parameter which is an instance of the target class.

As an illustrative example, we refer to the WordPress-Android app from the commit [2], we observe that the method `onDraw()` is moved from `GraphView` class to the *GraphViewContentView* class which makes the class `GraphView` more cohesive with an improvement in each of TCC and LCC from 0.2 to 0.5 and an improvement of LCOM from 32 to 28.

Finding 2. *Cohesion quality metrics, LCOM, TCC and LCC, tend to exhibit statistically significant variations with attribute and method-level moving-related refactoring operations. The refactorings that most influence cohesion are "Move Attribute", "Move Method" and "Extract And Move Method". However, LCOM tend to be more volatile under refactoring, which suggests Android developers to pay attention when dealing with cohesion.*

C. Results for Complexity Metrics

Reducing code complexity is one of the main challenges in any software system and one of the prominent goals of refactoring. We use the Weighted Methods per Class (WMC) to assess class complexity [17]. WMC for a given class is computed as the sum of the McCabe's cyclomatic complexity of its methods [32]. Being a direct metric, the higher WMC, the higher is the class complexity.

The distributions of the WMC metric depicted in Figure 3 and Table VI indicate a significant improvement after applying *Extract Super Class* refactoring by a median value of 10 with a medium effect size. Indeed, this refactoring operation is effective to remove code duplication and thus reducing complexity. Duplicate code often occurs when two classes perform similar tasks in the same way, or perform similar tasks in different ways. As a consequence, extracting a superclass can concentrate similar tasks and provide a built-in mechanism for simplifying such situations and removing duplicate code via inheritance. Moreover, various other refactorings tend to also improve WMC including *Move Method*, *Extract And Move Method*, *Pull Up Attribute*, *Extract Method*, and *Move Class* refactorings, but with less impact varying from 1 to 3 with negligible or small effect size. These improvements are expected since the applied refactoring operations deal with the simplification of methods inside a class. Particularly, the extraction of sub-methods that tend to break down long methods, or moving the methods to the appropriate class which decrease the complexity of the methods in the class.

An interesting example that shows the impact of Extract Method refactoring on complexity was found in the ownCloud app from commit [3] that involves the extraction of `readIsDeveloper()` method from `onCreate()` method in the `MainApp` class and the extraction of `showDeveloperItems()` method of `onCreate()` in the `Preferences` class. These refactorings reduce the complexity by decreasing the WMC metric from 7 to 3. Even though this commit is part of a pull request (# 13401), the developer tend to take care of the quality through refactoring commits.

Finding 3. *Several refactorings types tend to improve complexity by decreasing the WMC metric. The most impactful refactorings are "Extract Super Class", "Extract Method", and "Move Method" which typically help simplifying methods structure and/or reducing duplicate code.*

D. Results for Design Size Metrics

The design size is an indication of code density. We use two common metrics to estimate the size. The first is the Lines of Code (LOC) which counts the number of lines of code ignoring spaces and comments. The second metric is the Variables Quantity (VQTY) that counts the number of declared variables.

1) LOC: As shown in Figure 3 and Table VI, the refactorings *Extract Super Class*, *Pull Up Method*, *Push Down Method*, *Move Method*, and *Extract And Move Method* are the most influential refactorings that improve the LOC with a median ranging from 12 to 24 and a medium effect size. We notice that the size of the code elements significantly improve after the application of inheritance-related refactorings, as well as composing and moving method refactorings. For example, developers tend to apply *Extract Super Class* to reduce the size of its subclasses and reduce duplicated code, or *Extract And Move Method* to avoid code duplication and simplify the structure of the code.

2) VQTY: We observe from the results in Figure 3 and Table VI that VQTY is less impacted by refactoring than LOC with only 3 refactorings including *Move Attribute*, *Pull Up Method*, and *Push Down Method* which reduce significantly the Quantity variables metric with either negligible or small effect size. Similar to LOC, we speculate that moving and inheritance-related refactorings help reducing code duplication which will in turn reduce the number of declared variables in code fragments and/or improve code reuse. It is worth noting that our results match also with desktop applications [13], [20]. As an illustrative example, we refer to the WordPress-Android app, commit [4] which implements an *Extract Super Class*. The refactoring extracted the class `TagsFragment` from `PostSettingsTagsFragment` which clearly resulted in a reduced size.

Finding 4. *Most refactoring types tend to reduce the design size metrics LOC and VQTY. The most influential refactorings are Extract Super class, Pull UP/Push Down Method and Move Method which typically help reducing duplicate code, or moving code between classes, hence improving the design size.*

E. Inheritance

The inheritance is a key concept in any object-oriented (OO) programming infrastructure such as Android. Designing and implementing the inheritance relations in an Android app is expected to improve the overall quality of the app such as software reuse and extensibility. The depth of inheritance tree (DIT) is the most used metric to assess the inheritance in OO software applications.

We notice from Figure 3 and Table VI that five from all the applied refactorings do improve the DIT metric, including *Pull Up Attribute*, *Extract Super Class*, *Push Down Method*, *Pull Up Method*, and *Move Class*. The majority of these refactorings deal with changes applied to the class hierarchy. We expect that refactoring types that are mainly managing class inheritance do impact the DIT metric. A recent study showed that inheritance-related refactorings such as *Extract Super Class* and *Pull Up Method* tend to improve the depth of the inheritance to support software reusability and help in the elimination of code duplication [7]. Our qualitative analysis has shown scenarios of moving method down, from a super class, to a child class, for the purpose of sharing its behavior which is relevant only for some of its subclasses. One of the examples that show the inheritance improvement was found in the Nextcloud app, in commit ID [5]. Specifically, the developer applied a *Push Down Method* refactoring operations involving the class `AbstractIT` and its subclass `AbstractOnServerIT`. This was realized through pushing down the `after()`, `deleteAllFiles()`, `createDummyFiles()` and `waitForServer()` methods from `AbstractIT` to its subclass. These changes resulted in inheritance improvement for the `AbstractIT` class, with a drop of its DIT from 5 to 3.

Finding 5. *Hierarchy-level refactorings tend to improve the inheritance quality attribute (DIT). The most influential refactorings being "Pull Up Attribute", "Extract Super Class" and "Push Down Method". Improving the inheritance typically helps sharing common behavior across subclasses, reduces code duplication, and increases reusability which is a common practice in Android development.*

Looking at the control group results from Figure 4 and Table VI, we noticed that the different quality metrics did not exhibit any significant change with non-refactoring changes (control group), except for the LOC and VQTY metrics that tend to increase after each commit. Indeed, it is normal that the design size related metrics increase over time as the project evolves. These results provide more evidence that the metrics changes observed in the experiment data are due to refactoring activities and not to chance.

V. Implications and Discussions

A. Implications for researchers

Further exploit quality metrics and refactoring in mobile software development. The existing literature discusses different automatic refactoring approaches that help practitioners in detecting anti-patterns or code smells. More recently, Baqais and Alshayeb [11] show that there is an increase in the number of studies on automatic refactoring approaches and researchers have begun exploring how machine learning can be used in identifying refactoring opportunities. Since the features play a vital role in the quality of the obtained machine learning models, this study can help determine which metrics can be used as effective features in machine learning algorithms to accurately predict refactoring opportunities at different levels of granularity (*i.e.*, class, method, field), which can assist developers in automatically making their decisions. For example, using the most impactful metrics as a feature to predict whether a given piece of code should undergo a specific refactoring operation make developers more confident in accepting the recommended refactoring. Such knowledge is needed as, in practice, the built model should require as little data as possible.

B. *Implications for practitioners*

Android developers should be careful about their apps code quality. Our results indicate that developers can apply refactoring operations that do not improve their apps structural quality during refactoring, and particularly for the cohesion metric, LCOM. While LCOM tend to be very volatile under refactoring as also shown in prior works [38], [41], these results indicate that there is a risk that developers degrade their apps structural quality while performing refactoring changes. Given that Android apps should evolve quickly to add new user requirements, fix bugs or adapt to new technological changes, such refactorings may increase technical debt and thus cause developers to invest additional maintenance effort in the future in order to fix quality issues in their apps. Hence, developers need to pay attention to their refactoring edits.

Need for Android-specific refactoring tools. Our findings on the impact of refactoring on quality attributes/metrics can help build practical and customized refactoring recommendation tool for Android developers. For example, given the relatively small size and rapid evolution and release cycles of mobile apps, it is relevant to recommend refactoring opportunities for classes suffering from specific quality aspects, *e.g.*, coupling, complexity, etc.

C. *Implications for educators*

Learn refactoring best practices. Teaching the next generation of engineers best practices on refactoring and its impact on software quality in mobile apps and in software development, in general. Educators can use our study results and our dataset [44] to teach and motivate students to follow best refactoring practices while avoiding refactoring changes that may cause regression in their apps. In particular, our real world dataset of 42,181 refactorings from 300 Android apps, represents a valuable resource that could enable the introduction of refactoring to students using a "*learn by example*" methodology, illustrating best refactoring practices that should be followed and bad practices to be avoided.

VI. Threats to validity

Threat to internal validity. The accuracy of the refactoring detection tool, Refactoring Miner, can represent a threat to internal validity because it may miss the detection of some refactorings. However, previous studies report that Refactoring Miner has high precision and recall scores (98% and 87%, respectively) compared to other state-of-the-art refactoring detection tools [45], [51], which gives us confidence in using the tool. Furthermore, the CK-metrics tool could also have its own threats. While we conducted a manual inspection and double checked the values of the studied metrics with an alternative commercial tool, namely Scitools Understand, to make sure that the tool is reliable, still there could be errors that we did not notice. Another threat to internal validity could be related to the size of commit changes. In particular, the metric change in a given refactoring commit may or may not be related to the refactoring itself that occurred in that commit. To mitigate this problem, we adopted a widely-used causal inference method based on the Difference-in-Differences model that compare two groups, a treatment and a control group.

Threats to construct validity. A potential construct threat to validity could be related to the set of metrics being studied, as it may miss some properties of the selected internal quality attributes. To mitigate this threat, we select well-known metrics that cover various properties of each attribute, as reported in the literature [6], [17].

Threats to conclusion validity. Unlike other works on the impact of refactoring on quality metrics [6], [13], [15], [48], we employed the DiD method to compare the changes in quality metrics between a treatment and control group. Moreover, we used the non-parametric Wilcoxon rank-sum test and the Cliff's effect size, that do not make assumptions on the underlying data. As part of our future work, we plan to explore other quality aspects in mobile apps.

Threat to external validity. While we used a large sample of 300 open source Android apps written in Java, we cannot generalize our results to other open source or commercial mobile apps or to other technologies.

VII. Conclusion and Future Work

We presented a study aimed at investigating the impact of refactoring on quality metrics in Android apps. We mined 300 open-source apps containing 42,181 refactoring operations in total. We determined the effect each refactoring had upon the 10 chosen software quality metrics, and employed the difference-in-differences (DiD) model to determine the extent to which the metric changes brought about by refactoring differ from the metric changes in non-refactoring commits.

In one sense, our anticipated results were that the benefits of refactoring would be clearly reflected in the changes brought about in the software metrics. The observed results were not that simple however. For most refactoring type and metric combinations, the refactoring produced no significant change in the metric. On the other hand, some refactoring types yielded a broad improvement in several metric values. LCOM stood out as the least consistent metric, improving for some refactoring types and disimproving for others. For the non-refactoring commits, the metrics exhibit no significant change, other than (unsurprisingly) the design size metrics.

As future work, we plan to analyze other refactoring types and investigate their impact on internal and external quality attributes. We also plan to extend our study to more open source and commercial Android apps to better generalize our results, and to develop Android specific refactoring tools to better support developers during maintenance and evolution.

References

[1] Move Method, WordPress-Android https://github.com/wordpress-mobile/WordPress-Android/commit/ead8683e044a70fb3b288d562966c7ed442b8925. (Accessed on 05/02/2021).

[2] Move Method, WordPress-Android https://github.com/wordpress-mobile/WordPress-Android/commit/71a2b5277623415a7657accefc57c6599455aa3c. (Accessed on 05/02/2021).

[3] Extract Method, WordPress-Android https://github.com/owncloud/android/commit/7e460488f7fb2179c4476332dd5142a110450297. (Accessed on 05/02/2021).

[4] Extract Super Class, WordPress-Android https://github.com/wordpress-mobile/WordPress-Android/commit/3ff275dbf59b685666cc56ba8d094c1744955a5a. (Accessed on 05/02/2021).

[5] Push Down Method, NextCloud app https://github.com/nextcloud/android/commit/db6c1ba0554e5468ad568efe52c862c99ba7379c. (Accessed on 05/02/2021).

[6] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and M. Kessentini. On the impact of refactoring on the relationship between quality attributes and design metrics. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[7] E. A. AlOmar, P. T. Rodriguez, J. Bowman, T. Wang, B. Adepoju, K. Lopez, C. Newman, A. Ouni, and M. W. Mkaouer. How do developers refactor code to improve code reusability? In *International Conference on Software and Software Reuse*, pages 261–276. Springer, 2020.

[8] M. Alshayeb. Empirical investigation of refactoring effect on software quality. *Information and software technology*, 51(9):1319–1326, 2009.

[9] J. D. Angrist and J.-S. Pischke. *Mostly harmless econometrics: An empiricist's companion*. Princeton university press, 2008.

[10] M. Aniche. Ck-metrics. https://github.com/mauricioaniche/ck,, 2016. Accessed: 2020-01-09.

[11] A. A. B. Baqais and M. Alshayeb. Automatic software refactoring: a systematic literature review. *Software Quality Journal*, 28(2):459–502, 2020.

[12] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[13] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

[14] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 465–475, 2017.

[15] D. Cedrim, L. Sousa, A. Garcia, and R. Gheyi. Does refactoring improve software structural quality? a longitudinal study of 25 projects. In *30th Brazilian Symposium on Software Engineering*, pages 73–82, 2016.

[16] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia. How does refactoring affect internal quality attributes? a multi-project study. In *31st Brazilian Symposium on Software Engineering*, pages 74–83, 2017.

[17] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[18] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494, 1993.

[19] T. Das, M. Di Penta, and I. Malavolta. A quantitative and qualitative investigation of performance-related commits in android apps. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 443–447. IEEE, 2016.

[20] E. Fernandes, A. Chávez, A. Garcia, I. Ferreira, D. Cedrim, L. Sousa, and W. Oizumi. Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and Software Technology*, 126:106347, 2020.

[21] M. Fowler, K. Beck, J. Brant, W. Opdyke, and d. Roberts. *Refactoring: Improving the Design of Existing Code*. 1999.

[22] B. Geppert, A. Mockus, and F. Robler. Refactoring for changeability: A way to go? In *11th IEEE International Software Metrics Symposium*, page 10, 2005.

[23] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software engineering*, 31(10):897–910, 2005.

[24] G. Hecht, N. Moha, and R. Rouvoy. An empirical study of the performance impacts of android code smells. In *Proceedings of the international conference on mobile software engineering and systems*, pages 59–69, 2016.

[25] B. Joshi. Beginning solid principles and design patterns for asp.net developers. *Apress*, 2016.

[26] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 576–585. IEEE, 2002.

[27] M. Kessentini and A. Ouni. Detecting android smells using multi-objective genetic programming. In *IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 122–132, 2017.

[28] M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124. Springer, 1996.

[29] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimóthy, and N. Chrisochoides. Modeling class cohesion as mixtures of latent topics. In *IEEE International Conference on Software Maintenance*, pages 233–242, 2009.

[30] I. Malavolta, R. Verdecchia, B. Filipovic, M. Bruntink, and P. Lago. How maintainability issues of android apps evolve. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, 2018.

[31] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen. Understanding code smells in android applications. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 225–236. IEEE, 2016.

[32] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[33] R. Minelli and M. Lanza. Software analytics for mobile applications-insights and lessons learned, 2013.

[34] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni. Many-objective software remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):1–45, 2015.

[35] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol. Earmo: An energy-aware refactoring approach for mobile apps. *IEEE Transactions on Software Engineering*, 44(12):1176–1206, 2017.

[36] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In *IFIP Central and East European Conference on Software Engineering Techniques*, pages 252–266. Springer, 2007.

[37] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2011.

[38] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. Hemati Moghadam. Experimental assessment of software metrics using automated refactoring. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 49–58, 2012.

[39] E. Okike. A proposal for normalized lack of cohesion in method (lcom) metric using field experiment. *IJCSI International Journal of Computer Science Issues*, 7(4):19–26, 2010.

[40] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(3):1–53, 2016.

[41] M. Paixao, M. Harman, Y. Zhang, and Y. Yu. An empirical study of cohesion and coupling: Balancing optimization and disruption. *IEEE Transactions on Evolutionary Computation*, 22(3):394–414, 2017.

[42] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology*, 105:43–55, 2019.

[43] J. Pantiuchina, M. Lanza, and G. Bavota. Improving code: The (mis) perception of quality metrics. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 80–91, 2018.

[44] Replication package. https://github.com/stilab-ets/Android-refactoring, 2021.

[45] D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of github contributors. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 858–870, 2016.

[46] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In *Proceedings fifth european conference on software maintenance and reengineering*, pages 30–38. IEEE, 2001.

[47] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[48] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *International Workshop on Software Quality (WoSQ)*, pages 10–10, 2007.

[49] G. Szóke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy. Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 95–104. IEEE, 2014.

[50] L. Tahvildari and K. Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. In *European Conference on Software Maintenance and Reengineering*, pages 183–192. IEEE, 2003.

[51] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig. Accurate and efficient refactoring detection in commit history. In *IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 483–494. IEEE, 2018.

[52] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig. Accurate and efficient refactoring detection in commit history. In *International Conference on Software Engineering*, pages 483–494, 2018.

[53] A. Uchôa, C. Barbosa, W. Oizumi, P. Blenilio, R. Lima, A. Garcia, and C. Bezerra. How does modern code review impact software design degradation? an in-depth empirical study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 511–522. IEEE, 2020.

[54] F. Wilcoxon, S. Katti, and R. A. Wilcox. Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test. *Selected tables in mathematical statistics*, 1:171–259, 1970.

[55] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[56] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *IEEE/ACM international Conference on Automated software engineering*, pages 54–65, 2005.