On the Impact of Single and Co-occurrent Refactorings on Quality Attributes in Android Applications

Ali Ouni[a,*], Eman Abdullah AlOmar[b], Oumayma Hamdi[a], Mel Ó Cinnéide[c], Mohamed Wiem Mkaouer[d], Mohamed Aymen Saied[e]

[a]*Ecole de Technologie Superieure, University of Quebec, Montreal, QC, Canada*
[b]*Stevens Institute of Technology, Hoboken, NJ, USA*
[c]*University College Dublin, Dublin, Ireland*
[d]*Rochester Institute of Technology, Rochester, NY, USA*
[e]*Laval University, Quebec, QC, Canada*

Abstract

Android applications must evolve quickly to meet new user requirements, to facilitate bug fixing or to adapt to technological changes. This evolution can lead to various software quality problems that may hinder maintenance and further evolution. Code refactoring is a key practice that is employed to ensure that the intent of a code change is properly achieved without compromising internal software quality. While the impact of refactoring on software quality has been widely studied in object-oriented software in general, its impact in the specific context of mobile applications is still unclear. This paper reports on a large empirical study that aims to understand the impact of single and co-occurrent refactorings on quality metrics in Android applications. We analyze the evolution history of 800 open-source Android applications containing a total of 84,841 refactoring operations. We first analyze the impact of single refactoring operations on 21 common quality metrics using the Difference-in-Difference (DiD) statistical model. Then, we identify the most common co-occurrent refactorings using association rule mining, and investigate their impact on quality metrics using the DiD model. Our investigations deliver several important findings. Our results reveal that co-occurrent refactorings are quite prevalent in

*Corresponding author
Email address: ali.ouni@etsmtl.ca (Ali Ouni)

Android applications. Overall, 60% of the total number of refactoring commits contain multiple refactoring types, and 16 co-occurrent refactoring pairs tend to be applied together leading to a higher impact than single refactorings. We found that single refactorings have no statistically significant impact on quality metrics in 74.7% of the cases, a positive impact in 23.1% of the cases, and a negative impact in 2.2% of the cases. Whereas, co-occurrent refactorings have no statistically significant impact on quality metrics in 54.3% of the cases, a positive impact in 42.4% of the cases, and a negative impact in 3.3% of the cases. Our findings provide practical insights and suggest directions for researchers, practitioners, and tool builders to improve refactoring practices in the context of Android applications development.

Keywords: Mobile app, Android, refactoring, co-occurence, quality metrics, empirical study

---

1. Introduction

Mobile applications have to continuously evolve in order to cope with maintenance and evolution tasks. Often under such time pressure, poor design or implementation choices are made, which inevitably results in structural software quality problems [1–4].

Refactoring is a software engineering practice that aims to increase the quality of the code, making it easier to understand, modify and maintain. Measuring the impact of refactoring is challenging however, with several studies finding a poor correlation between refactoring and quality measures such as software metrics or code smells [5, 6]. Refactoring practices may present additional challenges in the context of Android applications due to their rapid evolution, short release deadlines, small code base, and heavy reuse of external libraries and code [7–9]. Indeed, mobile applications are inherently different to traditional desktop applications, *i.e.,* they run on mobile devices and are constrained by hardware specifics such as memory, CPU, varying screen sizes, etc. Unlike object-oriented software systems [6, 10–15], the impact of refactoring on quality metrics in mo-

bile applications has received little attention.

As a first attempt to address this problem, we conducted an empirical study [16] to investigate the impact of refactoring on quality metrics by analyzing the evolution history of 300 open-source Android applications containing a total of 42,181 refactoring operations. In particular, we analyzed the impact of these refactoring operations on 21 common quality metrics using a causal inference method based on the Difference-in-Differences (DiD) model [17]. The 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. However, the major limitation of this initial study is due to the presence of noise in the analyzed data, viz. (1) multiple refactoring operations are often applied in the same commit making the impact of a specific refactoring operation hard to measure and (2) commits with non-refactoring code changes are likely to affect quality metrics. Such code changes can add more noise to the analyzed quality metrics values, and impact the final observations.

To mitigate this problem, we conduct in this paper a large empirical study on a dataset composed of 800 open-source Android applications that are freely distributed in the Google Play Store. We analyze the impact of 18 commonly used refactoring operations on 21 well-known quality metrics in Android applications. We identified a total of 36,681 refactoring commits and 84,841 applied refactoring operations and measured quality metrics values before and after each refactoring operation. Then, we analyze these commits to distinguish between single and co-occurrent refactorings. To assess the impact of single refactorings, we investigated 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 [17]. For co-occurrent refactorings, we investigated commits containing multiple refactorings to discover patterns of co-occurrent refactorings that may occur in an Android application using association rule mining based on the Apriori algorithm [18] which is commonly used to find patterns in data. Finally, we analyze the impact

of these co-occurrent refactorings on quality metrics.

Our results reveal that co-occurrent refactorings are quite prevalent in Android applications. Overall, 60% of the total number of refactoring commits contain multiple refactoring types, and 16 co-occurrent refactoring pairs tend to be applied together leading to a higher impact than single refactorings. Results indicate that single refactorings have no statistically significant impact on quality metrics in 74.7% of the cases, a significant positive impact in 23.1% of the cases, and a significant negative impact in 2.2% of the cases. Whereas, Co-occurrent refactorings have no statistically significant impact on quality metrics in 54.3% of the cases, a significant positive impact in 42.4% of the cases, and a significant negative impact in 3.3% of the cases.

This paper is an extension of our earlier conference paper published in the Proceedings of the IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft 2021) [16]. It extends this earlier work in the following ways:

1. We extend our original dataset from 300 to 800 Open Source Android applications involving a total number of 84,841 applied refactoring operations,
2. We extend our study to explore both single and co-occurrent refactorings. Moreover, we extend the list of studied refactorings from 10 to 15 common refactoring operations,
3. We extend the list of quality metrics from 10 to 21 widely-used quality metrics,
4. We use a fine-grained impact analysis by investigating the impact of single and co-occurrent refactorings separately on quality metrics.

Replication package. We provide our comprehensive replication package available for future extensions and replications [19].

Paper organization. The remainder of this paper is organized as follows. Section 2 provides the necessary background in refactoring and quality metrics to understand our empirical study. Section 3 presents the design of our study,

while Section 4 presents and discusses our study results. Section 5 discusses the implications of our findings for researchers, practitioners and educators. Section 6 describes the threats to the validity of our study. Section 7 describes related work and finally, in Section 8, we summarize our conclusions and make suggestions for future work.

2. Background

This section provides the necessary background and concepts related to our study.

2.1. Refactoring

Refactoring is the process of reworking program source code without changing its functionality (*i.e.,* preserving its external behavior) to improve its structure in terms of readability, complexity, maintainability, extensibility, reusability, etc. The concept of refactoring was introduced by Opdyke [20] and popularized later in Fowler's well-known book [21]. Fowler first provided a comprehensive list of code smells along with a set of possible refactoring operations to fix each smell type. In this paper, we focus on 15 common refactoring operations, as described in Table 1.

Developers can refactor their code by applying single refactorings such as Extract Method or/and multiple refactorings, *i.e.,* co-occurrent refactorings, which are defined as set of two or more interrelated single refactorings applied to one or more code elements [22–24].

2.2. Internal Quality Attributes and Metrics

Internal quality attributes are indicators of code structural quality. In this paper, we analyze five internal quality attributes that are closely related to the 15 refactorings that we aim to investigate. Table 2 lists the five internal quality attributes analyzed including coupling, cohesion, complexity, design size and inheritance. These attributes are internal because they regard the internal code

Table 1: List of studied single refactoring operations.

Ref.	Refactoring	Description	Level
MM	Move Method	Moves a method from one class to another.	Method
EM	Extract Method	Creates a new method from an existing fragment of code.	Method
IM	Inline Method	Replaces calls to the method with the method's content and deletes the method itself.	Method
RM	Rename Method	Renames a 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
EC	Extract Class	Create a new class and moves fields and methods to this class.	Class
ESupC	Extract Superclass	Creates a superclass from two classes with common attributes and methods.	Class
ESubC	Extract Subclass	Creates a subclass and moves features that are used only in certain cases to this class.	Class
RC	Rename Class	Renames a class.	Class
MC	Move Class	Moves a class to another package.	Class
EI	Extract Interface	Extracts a new general interface from two or more existing classes.	Interface

structure including class hierarchies and compositions in methods and fields [25]. Each quality attribute can be measured by one or multiple quality metrics.

Quality metrics are widely-used to quantify software quality. One of the commonly used metric suites is the CK suite, a quality metric catalog defined by Chidamber and Kemerer [26]. Table 2 presents, for each quality attribute considered in this paper, its associated quality metrics and also a description of each metric. Special mention must be made of the LCOM metric. Although LCOM is a cohesion metric, and cohesion is a positive properly, LCOM measures *lack* of cohesion so it increases when cohesion deteriorates and decreases when cohesion improves. We have taken care to ensure that this does not cause confusion in reporting results related to the LCOM metric.

3. Empirical Study Design

In this paper, we extend our previous study [16] while ensuring that we have high quality-data by removing noise to enhance data analysis and deliver sound data-driven conclusions. In this section, we describe our empirical study design following the guidelines from Runeson et al. [31] including our research questions, and the experimental setup.

Table 2: The list of studied quality attributes and metrics.

Quality Attribute	Associated Metrics	Metric Description
Coupling	Coupling Between Objects (CBO)	Number of classes that are coupled to a particular class [26].
	Number Of Static Invocations (NOSI)	Number of invocations of static methods [27]
	Response For a Class (RFC)	Number of methods invoked in the body of a class [26].
	Fan-in (FANIN)	The number of other classes that reference a class [25].
	Fan-out (FANOUT)	The number of other classes referenced by a class [25].
Cohesion	Lack Of Cohesion of Methods (LCOM)	Numbers of pairs of methods that have no instance variable references in common [26].
	Tight Class Cohesion (TCC)	Number of directly connected public methods in a class [27].
Complexity	Cyclomatic Complexity (CC)	Measure of the complexity of a module's decision structure [28].
	Weighted Methods per Class (WMC)	The sum of all the complexities of the methods (McCabe's Cyclomatic Complexity) in the class [26].
	Essential Complexity (Evg)	Measure of the degree to which a module contains unstructured constructs [28].
	Nesting (MaxNest)	Maximum nesting level of control constructs [29].
Design Size	Lines of code (LOC)	Number of lines of code ignoring spaces and comments [27].
	Blank Line of Code (BLOC)	Number of Blank lines [25].
	Lines with Comments (CLOC)	Number of lines containing comment [25].
	Statements (STMTC)	Number of statements [25].
	Instance Variables (NIV)	Number of instance variables [25].
	Instance Methods (NIM)	Number of instance methods [25].
	Variable Quantity (VQTY)	Number of declared variables [27].
Inheritance	Depth of Inheritance Tree (DIT)	Number of classes that a particular class inherits from [26].
	Number Of Children (NOC)	The number of direct descendants (subclasses) of a class [26].
	base Classes (IFANIN)	Number of immediate base classes [30].

3.1. Research questions

Our study aims at addressing the following research questions.

- **RQ1**. **How do single refactoring operations applied by Android developers affect quality metrics?** This first research question aims at investigating how a single refactoring operation can affect quality metrics in Android applications.

- **RQ2**. **What are the most common co-occurrent refactorings applied by Android developers?** With this research question, we aim at identifying the different co-occurrent refactorings that are commonly applied in Android applications.

- **RQ3**. **How do co-occurrent refactorings affect quality metrics?** The goal of this research question is to assess how co-occurrent refactorings identified in RQ2 affects quality metrics.

3.2. Empirical Study Setup

To address our research questions, we design a controlled experiment where we select two groups of code changes. A first group that consists of refactoring-

related code changes (*i.e.,* treatment group), and a second group that consists of 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 eight main steps: (1) Android applications selection, (2) refactoring extraction, (3) commit extraction, (4) non-refactoring changes extraction, (5) quality metrics measurement, (6) single refactorings impact analysis, (7) co-occurrent refactorings identification, and (8) co-occurrent refactorings impact analysis.

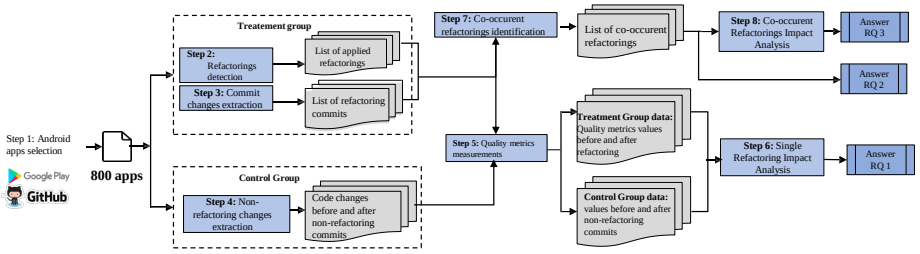

Figure 1: The overall process of our empirical study.

3.2.1. Step 1: Android applications selection

We target open-source Android applications that are freely distributed in 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 applications. Thereafter, inspired by previous works [16, 32, 33], we applied the following filters to exclude:

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

Thereafter, we randomly selected a set of 800 applications which represents over 40% of the final set, exhibiting a total of 84,841 refactoring operations. We focused our study to this set of applications for computational reasons. It is worth noting that the sample size of 800 applications and 84,841 refactoring operations is larger than related studies on the impact of refactoring on software quality [6, 12, 13, 16], and than typical samples in software engineering research [34]. Overall, the sample of selected applications have a median size of 6,854 LOC, a median number of contributors of 2, and a median number of forks of 3. Table 3 and Figure 2 summarize the statistics about the collected dataset.

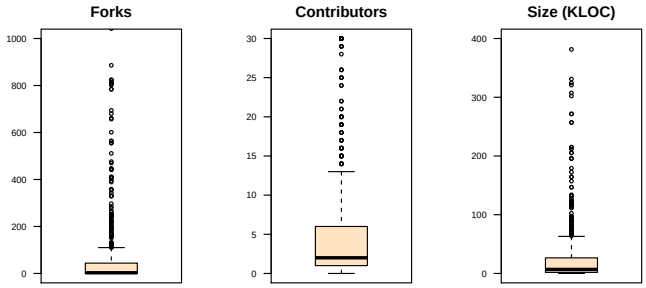

Figure 2: Dataset statistics in the terms of the number of forks, contributors, and size.

Table 3: Dataset statistics.

Statistic	**Count**
# of Android applications	800
# of commits with refactorings	36,681
# of commits containing a single refactoring type	14,672
# of commits containing multiple refactoring types	22,009
# of commits without refactorings	36,681
# of refactoring operations	84,841
Total number of commits	723,360

3.2.2. Step 2: Refactoring detection

In this step, we collect all the refactoring operations applied to the studied applications. We utilize RefactoringMiner (version 2.1.0) [35] 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 [36] algorithm for differencing object-oriented models. RefactoringMiner has been shown to achieve a precision of 98% and a recall of 87% [35, 37]. 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 single refactoring types detected by RefactoringMiner were considered in this study, except the Rename Method and Rename Class refactorings for RQ1 as they are not directly related to any of the structural metrics considered in our study. Overall, our extraction process identifies a list of 15 common single refactoring types which are amongst the most common refactoring types [10, 13, 15, 38–40]. Tables 1 and 4 report the list and the number of refactorings, respectively, that are investigated in our study.

Table 4: The list of refactoring applied to the analyzed applications.

Refactoring type	Number
Extract Method	20,494
Rename Method	13,872
Move Method	11,496
Move Attribute	10,303
Push Down Method	6,748
Inline Method	5,123
Pull Up Attribute	3,374
Pull Up Method	2,246
Extract Interface	2,160
Extract Class	2,021
Extract Sub Class	1,930
Push Down Attribute	1,910
Move Class	1,200
Extract Super Class	1,140
Rename Class	924
Total	**84,841**

3.2.3. 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[1] 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.

3.2.4. 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 an equivalent set of non-refactoring commits representing our control group in such a way that the number of non-refactoring commits is equal to the number of refactoring commits for each project. For each commit, we collected its ID as well as the commit that precede it. Thereafter, we performed the same procedure adopted in Step 3 to collect their source code.

3.2.5. Step 5: Quality metrics measurement

We selected a comprehensive set of commonly acknowledged software quality metrics based on prior works [10, 16, 26, 28, 29, 41–45]. Then, we checked if the metrics assess several object-oriented design aspects to map each internal quality attribute to the appropriate structural metric(s). More generally, we extract, from the literature review, all the associations between metrics with internal quality attributes.

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

[1] `https://docs.github.com/en/rest`

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 use two widely used software quality software metrics suite tools, the CK-metrics[2] tool and the Scitools Understand[3] tool including CK suite, McCabe, and Lorenz and Kidd's metrics. CK-metrics is a command-line based tool provided by Aniche [27]. Understand is a popular quality assurance and static analysis framework that allows calculating a variety of quality metrics using command line. We considered a total of 21 common quality metrics that cover five different quality attributes including coupling, cohesion, complexity, design size, and inheritance. The list of considered quality metrics is provided in Table 2.

3.2.6. Step 6: Single 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 [46], 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) [47]

[2] `https://github.com/mauricioaniche/ck`

[3] `https://www.scitools.com/`

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

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 [17]. In econometrics, difference-in-differences (DiD) method is 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 3, 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:

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

where $Y^R_{M_i}$ is the median perceived impact after the application of the set of refactorings of type 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.

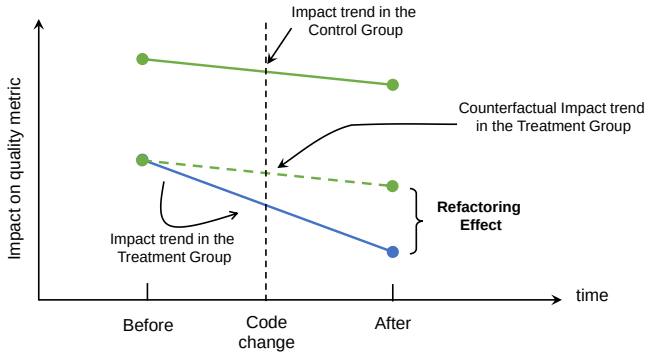

Figure 3: An example of the causal inference method using a DiD model showing the refactoring impact on a quality metric before versus after the application of refactoring [16].

3.2.7. Step 7: Co-occurrent Refactorings Detection

After the extraction of all refactoring operations (Step 2), we employ association rule mining using the *Apriori* algorithm [18]. It has been successfully used to mine association between items in many problems such as market basket analysis [48], intrusion detection [49] and supply chain management [50] . The algorithm parses the dataset, *i.e.,* transactions, and generates frequent itemsets based on filtering criteria set. Association rules are generated during searching for frequent itemsets. An association rule is defined as an implication of the form X $\Rightarrow$ Y, where X, Y $\subseteq$ I and X $\cap$ Y $= \emptyset$.

Let $I = \{i_1, i_2, ..., i_n\}$ be a set of n items, and $T = \{t_1, t_2, ..., t_m\}$ a set of m transactions. In our study, T is the set of classes present in version, and each item in the set I indicates the presence of two specific refactoring types. Therefore, an association rule translates a co-occurrence between a refactoring R_i and other refactoring R_j on the same class. Specifically, the association rule is written as follows: $\mathit{refactoring}(R_i) \Rightarrow \mathit{refactoring}(R_j)$.

We use the support [18], confidence [18] and lift [51] scores to quantify the

degree of association between each pair of refactorings.

1. *Support*: is an indication of how frequently an itemset appears in the dataset and consists of the proportion of transactions in the dataset that contain both R_i and R_j.

$$Support(R_i \Rightarrow R_j) = P(R_i \cup R_j) \tag{2}$$

2. *Confidence*: is the proportion of transactions in the dataset containing R_i, that also contain R_j.

$$Confidence(R_i \Rightarrow R_j) = P(R_i \cup R_j)/P(R_i) \tag{3}$$

3. *Lift*: is the ratio of the observed support to that expected if R_i and R_j were independent.

$$Lift(R_i \Rightarrow R_j) = P(R_i \cup R_j)/(P(R_i) \times P(B)) \tag{4}$$

The range of values for support and confidence is between 0 and 1, whereas lift can take any value between 0 and $+\infty$. When the lift value is greater than 1, it implies that the refactoring pair is highly correlated.

Moreover, we use the Pearson's Chi-square coefficient and Cramer's V [52] tests to determine if there were significant associations between refactoring operations. Specifically, for any Chi-square test that was found to be significant (p-value < 0.001), Cramer's V test is calculated and it has a value between 0 and 1. A value of 0 indicates complete independence, and a value of 1 indicates complete association. The formula is given in Equation 5:

$$V = \sqrt{\frac{\chi^2}{n \times min(row - 1, col - 1)}} \tag{5}$$

3.2.8. Step 8: Co-occurrent Refactorings Impact Analysis

In this step, we evaluate the impact of co-occurrent refactorings on internal quality attributes for each class in which c-occurrent refactorings was applied. We chose to perform the analysis in a class-level due to a recent study about the

effect of co-occurrent refactorings on code smells indicating that co-occurrent refactorings are often applied at class level [53]. To do so, we perform the same statistical method employed in Step 6 using the DiD model.

4. Empirical Study Results

This section describes and discusses the results of our investigations.

4.1. RQ1. How do single refactoring operations applied by Android developers affect quality metrics?

In this research question, we assess the impact of single refactoring operations on quality metrics. In particular, we aim at understanding how single refactorings affect each quality metric. Table 5 provides the detailed results where each column reports *(i)* the impact of the corresponding 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). Overall, Table 5 reports 273 measurements on the experimental group, *i.e.,* 13 refactoring types $\times$ 21 quality metrics. The results indicate that in 74.7% of the cases (204 out of 273 measurements as represented in the while color in the table), refactoring have no statistically significant impact on quality metrics, while it has a positive impact in 23.1% of the cases (63 out of 273 as presented in blue) and a negative impact in 2.2% of the cases (6 out of 273 as presented in red). In the following, we present and discuss the obtained results for each quality attribute along with illustrative examples from our experiments.

Finding 1.1. *Single refactorings have no statistically significant impact on quality metrics in 74.7% of the cases, a significant positive impact in 23.1% of the cases, and a significant negative impact in 2.2% of the cases.*

Table 5: The impact of refactoring (treatment group) and non-refactoring (control group) changes on quality metrics.

Data	Change	Measure	Cohesion		Coupling					Comlexity				Inheritance			Design Size						
			LCOM	TCC	CBO	RFC	FANIN	FANOUT	NOSI	WMC	EVG	MaxNest	CC	DIT	NOC	IFANIN	LOC	BLOC	CLOC	STMTC	NIM	NIV	VQTY
Treatment Group	Extract Method	Refactoring Impact	2	0	0	0	0	0	0	-3	-2	-1	-1	0	0	0	7	0	0	15	2	-6	0
		Behavior	↑	-	-	-	-	-	-	↓	↓	↓	↓	-	-	-	↑	-	-	↑	↑	↓	-
		P-value (δ)	0.77(L)	<0.05(N)	0.05 (N)	0.05 (N)	0.79 (N)	0.79 (N)	<0.05 (N)	<0.05(N)	0.11 (N)	<0.05(N)	<0.05(N)	0.11 (N)	0.19 (S)	<0.05 (N)	0.47 (M)	0.06 (N)	<0.05 (N)	0.54 (M)	0.09 (N)	0.41 (M)	<0.05(N)
	Move Method	Refactoring Impact	0	0	0	-4	-1	-2	0	0	-2	-1	0	-1	0	0	-2	0	0	-5	-1	-2	0
		Behavior	-	-	-	↓	↓	↓	-	-	↓	↓	-	↓	-	-	↓	-	-	↓	↓	↓	-
		P-value (δ)	<0.05(N)	0.19 (S)	<0.05 (N)	<0.05 (N)	<0.05(N)	<0.05(S)	<0.05(N)	<0.05(N)	<0.05(N)	0.37 (M)	<0.05 (N)	0.45 (S)	0.24 (S)	<0.05(N)	<0.05(S)	0.07 (N)	<0.05(N)	0.68 (S)	<0.05(N)	0.64 (N)	0.29 (N)
	Move Attribute	Refactoring Impact	0	0	0	-6	-1	-3	0	-2	0	-2	0	-2	0	0	-3	1	0	-5	1	-7	0
		Behavior	-	-	-	↓	↓	↓	-	↓	-	↓	-	↓	-	-	↓	↑	-	↓	↑	↓	-
		P-value (δ)	<0.05(N)	0.12 (N)	<0.05(N)	<0.05(N)	<0.05(N)	<0.05(S)	<0.05(N)	<0.05(N)	<0.05(N)	0.41 (M)	0.06 (N)	0.49 (S)	0.22 (S)	<0.05(N)	<0.05(S)	0.08 (N)	<0.05(N)	<0.05(N)	<0.05(N)	0.59 (L)	0.35 (N)
	Inline Method	Refactoring Impact	-2	0	0	-4	-1	-1	0	-2	-4	-1	0	-3	0	0	-4	1	0	6	-2	0	-8
		Behavior	↓	-	-	↓	↓	↓	-	↓	↓	↓	-	↓	-	-	↓	↑	-	↑	↓	-	↓
		P-value (δ)	0.10 (N)	0.54 (N)	<0.05(N)	<0.05(S)	<0.05(N)	<0.05(N)	<0.05(N)	<0.05(N)	0.47 (S)	0.36 (M)	0.07 (N)	<0.05(N)	0.31 (S)	<0.05(N)	0.49 (L)	0.16 (S)	<0.05(N)	0.87 (L)	0.08 (N)	0.78 (N)	<0.05(M)
	Pull Up Method	Refactoring Impact	-7	0	-1	-5	0	0	0	-3	0	-2	0	-3	0	0	-6	0	0	2	-2	-2	-2
		Behavior	↓	-	↓	↓	-	-	-	↓	-	↓	-	↓	-	-	↓	-	-	↑	↓	↓	↓
		P-value (δ)	<0.05(S)	0.57 (N)	0.11 (N)	<0.05(N)	0.11 (1)	1(N)	<0.05 (N)	<0.05(M)	0.75 (N)	0.65 (N)	0.16 (N)	0.51 (L)	0.08 (N)	0.11 (N)	0.56 (L)	0.06 (N)	<0.05(N)	0.80 (L)	0.18 (S)	0.86 (M)	<0.05(N)
	Pull Up Attribute	Refactoring Impact	-9	0	-2	-1	0	0	0	-4	-3	-1	0	-1	0	0	0	0	0	-3	-3	-1	-1
		Behavior	↓	-	↓	↓	-	-	-	↓	↓	↓	-	↓	-	-	-	-	-	↓	↓	↓	↓
		P-value (δ)	0.21 (S)	<0.05(N)	0.10 (N)	0.15 (S)	<0.05(N)	<0.05(N)	<0.05(N)	0.16 (S)	0.15 (S)	0.15 (S)	0.24 (N)	0.35 (M)	<0.05(N)	<0.05(N)	<0.05(N)	<0.05(N)	0.06 (N)	0.15 (S)	0.15 (S)	0.21 (N)	<0.05(N)
	Extract Class	Refactoring Impact	-4	0	-2	-1	0	0	0	-8	-1	1	-3	0	0	0	-1	0	0	-3	-2	-1	-2
		Behavior	↓	-	↓	↓	-	-	-	↓	↓	↑	↓	-	-	-	↓	-	-	↓	↓	↓	↓
		P-value (δ)	0.10 (N)	<0.05(N)	0.14 (N)	<0.05(N)	1(N)	1(N)	<0.05(N)	<0.05(N)	<0.05(N)	0.12 (N)	<0.05 (S)	0.64 (N)	<0.05(N)	<0.05(N)	<0.05 (N)	<0.05(N)	<0.05(N)	0.06 (N)	0.11 (N)	0.11 (N)	<0.05(N)
	Push Down Method	Refactoring Impact	-1	0	1	0	0	2	0	-1	-1	-1	0	-4	1	1	-5	0	0	-2	2	2	0
		Behavior	↓	-	↑	-	-	↑	-	↓	↓	↓	-	↓	↑	↑	↓	-	-	↓	↑	↑	-
		P-value (δ)	0.14 (N)	0(N)	0.06 (N)	0.13(N)	0.31 (N)	0.09 (N)	0.16 (N)	<0.05(N)	<0.05 (N)	<0.05(N)	0.39 (N)	<0.05(N)	<0.05 (N)	<0.05(N)	<0.05 (N)	<0.05 (N)	0.13 (N)	0.19 (N)	0.16 (S)	0.21 (S)	<0.05 (N)
	Extract Super Class	Refactoring Impact	-10	0	-1	-6	0	0	0	-3	-1	-5	-1	-3	0	0	-5	-1	0	1	-1	-2	-1
		Behavior	↓	-	↓	↓	-	-	-	↓	↓	↓	↓	↓	-	-	↓	↓	-	↑	↓	↓	↓
		P-value (δ)	<0.05(N)	0.45 (N)	<0.05(N)	<0.05(S)	0.16 (N)	0.13 (N)	0.15 (N)	<0.05(S)	0.11 (N)	<0.05(M)	0.12 (N)	0.44 (N)	0.06 (N)	0.06 (N)	<0.05(M)	0.50 (N)	<0.05 (N)	0.71 (N)	0.10 (N)	<0.05(N)	<0.05 (N)
	Extract Sub Class	Refactoring Impact	-14	0	1	2	-1	-1	0	0	0	0	0	0	2	0	0	0	-1	-3	3	-1	-1
		Behavior	↓	-	↑	↑	↓	↓	-	-	-	-	-	-	↑	-	-	-	↓	↓	↑	↓	↓
		P-value (δ)	<0.05(S)	0.15 (N)	0.65(N)	0.21 (N)	0.15 (N)	0.06 (N)	0.29 (N)	0.05 (N)	0.14 (N)	1 (N)	0.07 (N)	0.25 (N)	0.74 (N)	0.12 (N)	0.18 (N)	<0.05(N)	0.31 (N)	0.07(N)	0.06 (N)	<0.05(N)	<0.05 (N)
	Extract Interface	Refactoring Impact	-2	0	0	-7	0	0	0	-6	3	-1	0	0	0	0	-6	0	0	-3	-2	-2	-3
		Behavior	↓	-	-	↓	-	-	-	↓	↑	↓	-	-	-	-	↓	-	-	↓	↓	↓	↓
		P-value (δ)	0.07 (N)	0.81 (N)	0.05 (N)	<0.05 (N)	1 (N)	1(N)	0.07 (N)	<0.05(N)	0.06 (N)	<0.05 (N)	0.11 (N)	0.10 (N)	0.43 (M)	0.24 (N)	<0.05 (N)	0.08 (N)	<0.05(N)	0.96 (N)	0.76 (N)	0.76 (S)	<0.05 (S)
	Move Class	Refactoring Impact	0	0	1	-6	1	-2	0	-2	-1	-1	0	-1	0	0	-3	-1	-1	0	1	0	0
		Behavior	-	-	↑	↓	↑	↓	-	↓	↓	↓	-	↓	-	-	↓	↓	↓	-	↑	-	-
		P-value (δ)	<0.05 (N)	0.16 (N)	0.20 (N)	<0.05 (S)	<0.05(N)	<0.05 (N)	<0.05(N)	<0.05 (N)	0.62 (N)	<0.05(N)	0.08 (N)	0.46 (N)	0.31 (N)	<0.05 (N)	<0.05(S)	0.16 (N)	<0.05 (N)	0.65 (N)	<0.05 (N)	0.56 (N)	0.27 (N)
	Push Down Attribute	Refactoring Impact	-11	0	1	-4	0	-1	0	-4	-1	-1	-	-4	2	1	0	0	0	0	0	0	-1
		Behavior	↓	-	↑	↓	-	↓	-	↓	↓	↓	0	↓	↑	-	-	-	-	-	-	-	↓
		P-value (δ)	<0.05(M)	1(N)	0.41 (N)	<0.05(N)	0.87 (N)	0.66 (N)	0.09 (N)	<0.05(N)	0.07 (N)	0.43 (N)	1 (N)	<0.05(N)	<0.05 (N)	0.14 (S)	0.10 (N)	0.79 (M)	0.05 (N)	0.44 (N)	0.32 (N)	0.46 (N)	<0.05 (N)
Control Group	Non-refactoring Code change	Non-refactoring Impact	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	0	0	0	0	0	-2
		Behavior	-	-	-	-	-	-	-	-	-	-	-	-	-	-	↑	-	-	-	-	-	↓
		P-value (δ)	0.18 (N)	0.17 (N)	0.11 (N)	0.08 (N)	1(N)	1(N)	0.06 (N)	0.16 (N)	0.76 (N)	0.18 (N)	0.09 (N)	0.32 (N)	0.15 (N)	0.30 (N)	<0.05 (N)	0.79 (N)	0.06 (N)	0.26 (N)	0.06 (N)	<0.05 (N)	<0.05 (S)

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.

4.1.1. Results for Coupling

- **CBO:** From Table 5, we observe that *Extract Super Class* is the only influential refactoring that improves the CBO by a median value of 1, with a negligible effect size. We expect that the *Extract Super Class* refactoring has a direct impact only on size and inheritance (See Table 5). Hence, if there are two different classes that implement similar functionalities, *Extract Super Class* can be applied to create a single Super Class that implements such common functionalities. Consequently, the duplicate code will be removed from both subclasses, thereby reducing their size and increasing the inheritance depth. Still, it can indirectly reduce dependencies through polymorphism [54, 55]. We also observe that other inheritance-related refactorings such as *Extract Sub Class*, *Push Down Method*, *Push Down Attribute* tend to have a negative impact on CBO. While, the obtained results for those refactoring are not statistically different from the control group, this finding indicates that developers may need to pay attention when dealing with inheritance aspects to avoid increasing the coupling in their code.

- **RFC:** As can be seen from Table 5, *Extract Interface* is the most impactful refactoring which improves RFC by 7, with a negligible effect size. Moreover, *Move Class*, *Extract Super Class*, *Move Attribute* and *Pull Up Method* have shown to improve RFC by a median score between 7 and 5 with a negligible and small effect size, while less impact is observed by *Move Method*, *Inline Method*, *Push Down Attribute* and *Extract Class* refactorings with a median score between 4 and 1 with a negligible and small effect size.

- **FANIN:** From Table 5, we observe that *Move Method*, *Move Attribute*, and *Inline Method* are the most influential refactorings which improves FANIN by 1 with a negligible effect size. Furthermore, *Move Class* tend to deteriorate FANIN by 1 but with a negligible effect size.

- **FANOUT:** From Table 5, we observe that *Move Attribute* is the most influential refactoring on FANOUT which improve it by 3. Moreover, *Move Method* and *Move Class* refactorings have shown to improve FANIN by a median score of 2 with a negligible and small effect sizes. Whereas, *Inline Method* tend to have less impact on the metric with a median improvement of 1 and a negligible effect size.

- **NOSI:** From Table 5, 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 result 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 applications share some similarities with desktop applications in terms of coupling metrics which generally tend to be positively impacted after applying refactoring [56, 57]. Hence, we found that the *Extract Method*, *Move Method* and *Inline Method* refactorings are the most applied refactorings to improve coupling in Android applications, similarly to desktop applications [6, 25, 58].

As an illustrative example, we refer to the HIITMe[4] app, Commit #e8b345b [59] which implements a *Move And Inline Method*. This refactoring moves the `startDrag()` method from the `ScrollingProgramView` class to `ProgramDetailView`. Then, the method was inlined with the `startDrag(DraggableView, int, int)` method which clearly resulted in a reduced coupling with a drop of the CBO from 11 to 8, the RFC from 12 to 10, the FANIN from 4 to 2. However, the refactoring produced no change in the FANOUT and NOSI metrics.

[4] `https://github.com/AlexGilleran/HIITMe`

Finding 1.2. *Refactoring has a significant positive impact on coupling in terms of the CBO, RFC, FANIN and FANOUT metrics with a negligible or small effect size*, while no significant impact was found on the NOSI metric.The most influential refactorings that promote low coupling are *Move Attribute*, *Move Method*, *Inline Method* and *Extract Super Class*.

4.1.2. Results for Cohesion

- **LCOM:** The results from Table 5 show that LCOM is improved with various refactoring types. The most influential ones are *Extract Sub Class*, *Push Down Attribute* and *Extract Super Class* as they decrease the median value by 14 (small effect size), 11 (negligible effect size) and 10 (medium effect size), respectively. Whereas, *Pull Up Method* tend to have less impact on LCOM with a median improvement of 7 (small effect size). These results are expected because Pull Up, Push Down and Move operations are typically recommended for moving code elements across classes [60]; thus, the class cohesion should improve.

- **TCC:** From Table 5, we observe that the TCC metric has not been impacted by any of the applied refactorings since when comparing the values before and after refactoring, no statistically significant difference is observed. This is an unexpected result since cohesion is generally expected to be improved after applying the moving-related refactoring operations [60]. However, it is worth noting that none of the refactorings applied deteriorate the metric.

Overall, we notice that we found a number of similarities between our results for Android applications and prior works on desktop applications. Similar to Fernandes et al. [58] and Chavez et al. [25], we found that *Extract SubClass* and *Extract SuperClass* improve the cohesion. One of the examples that shows the impact of *Extract SubClass* refactoring on cohesion was found in the Glide Player[5] application from Commit #c8a6d71 [61] that involves the extraction

[5] `https://github.com/philn/glide`

of `MusicLibraryFragment` subclass from the `LibraryFragment` class. This refactoring improved the cohesion by decreasing the LCOM metric from 77 to 71. We also observe a slight increase in terms of TCC by 0.1.

Finding 1.3. *Some refactorings have a significant positive impact on cohesion in terms of the LCOM metric, while no significant impact was found on the TCC metric. The most influential refactorings that promote low cohesion are "Extract Sub Class", "Extract Super Class", and "Push Down Attribute" with an effect size ranging from negligible to medium.*

4.1.3. Results for Complexity

- **WMC:** The results of the WMC metric depicted in Table 5 indicate that *Extract Class* and *Extract Interface* are the most influential refactorings that improve the WMC with a median value of 8 and 6, respectively, with a negligible effect size in both cases. Indeed, the extraction-related refactoring operations are effective at removing 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 [21]. Moreover, several other refactorings tend to also improve WMC including *Extract Method*, *Inline Method*, *Move Attribute*, *Move Class*, *Push Down Attribute*, *Pull Up Attribute*, *Pull Up Method*, *Extract Super Class* and *Push Down Method* refactorings, but with less impact varying from 1 to 4 with negligible and small effect size. These improvements make sense 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.

- **EVG:** As can be seen in Table 5, *Move Method* is the most influential refactoring on EVG which improve it by 2 with a negligible effect size. *Extract Class* and *Push Down Method* have shown to improve EVG by a median score of 1 with a negligible effect size, each.

- **MaxNest:** The results of the MaxNest metric depicted in Table 5 indicate a significant improvement when applying *Extract Super Class* refactoring with a median value of 5 exhibiting a medium effect size. Moreover, less impact is observed for *Extract Method*, *Push Down Method*, *Extract Interface* and *Move Class* with a negligible effect size, in each refactoring.

- **CC:** As it can be seen from Table 5, *Extract Class* is the most influential refactoring that improve CC with a median value that is significantly decreased by 3, even though it is accompanied by a small effect size. Moreover, we observe that *Extract Method* improves CC with a median of 1 with a negligible effect size.

It is worth noting that our findings in Android applications share some similarities with desktop applications for the complexity which is improved by applying *Extract Method* in both Android and desktop applications [58].

One of the examples that show the complexity improvement was found in the Daily Dozen Android[6] app, in Commit #8aa4b45 [62]. Specifically, the developer applied an *Extract Class* refactoring operation. This was realized through extracting the `DateFragment` class from the `MainActivity` class. This change resulted in an important complexity improvement for the `DateFragment` class, with a drop of its WMC from 10 to 6, its EVG from 4 to 3 and its CC and MaxNest from 3 to 1, each.

Finding 1.4. *Several refactoring types tend to improve complexity by decreasing the WMC, EVG, MaxNest and CC metrics with a effect size ranging from negligible to medium.* The most impactful refactorings are related to code extraction such as "Extract Method", "Extract Class", "Extract Interface" and "Extract Super Class" which typically help simplifying methods structure and/or reducing duplicate code.

[6] `https://github.com/nutritionfactsorg/daily-dozen-android`

4.1.4. Results for Inheritance

- **DIT:** We notice from Table 5 that *Push Down Method* and *Push Down Attribute* improve DIT by a median value of 4 with a negligible effect size, each.

- **NOC:** We observe from the results in Table 5 that the *Push Down Method* and *Push Down Attribute* increase the NOC metric by a median of 1 and 2 with a negligible effect size. The pushing down refactorings are typically applied when a method is needed by one or more subclasses, but not all of them. It could be useful to create an intermediate subclass and move the method to it. Therefore, this will allow the increase of immediate subclasses of a class.

- **IFANIN:** As can been seen in Table 5, applying *Push Down Method* and *Push Down Attribute* tend to increase IFANIN by a median value of 1 with a negligible effect size, each. However, this result is contradictory since pushing down methods do not directly impact on the number of classes.

One of the examples that illustrates the inheritance improvement was found in the History Cleaner Pro[7] app, in Commit # 23852db [63]. Specifically, the developer applied a *Push Down Method* refactoring involving the class `CleanItem` and its subclass `CleanItemStub`. This was realized through pushing down the `getDataPath()`, `getIcon()`, and `getUniqueId()` methods from the class `CleanItem` to `CleanItemStub` subclass. These changes resulted in inheritance improvement for the `CleanItem` class, with a drop of its DIT from 6 to 2.

Finding 1.5. *Refactoring has a significant positive impact on inheritance in terms of the DIT metric, while a negative impact was found on the NOC and IFANIN metrics. The most influential refactorings that promote low inheritance are "Push Down Method", and "Push Down Attribute" with a negligible effect size.*

[7] `https://github.com/JohnNPhillips/HistoryCleanerPro`

4.1.5. Results for Design Size

- **LOC:** As shown in Table 5, the refactoring *Extract Interface* is the most influential refactoring that improves the LOC metric by 6 with a negligible effect size. Moreover, several other refactorings such as *Extract Super Class* and *Push Down Method* have shown to increase the LOC with a median of 5 and a negligible effect size, while less impact is observed by the *Move Attribute*, *Move Method* and *Extract Class* with a median value varying from 1 to 3 with an effect size ranging from medium to small.

- **BLOC:** From Table 5, we observe that the BLOC metric has not been impacted by any of the applied refactorings since when measuring the refactoring impact, no statistically significant result is observed.

- **CLOC:** We observed also that CLOC metric has not been impacted by any of the applied refactorings. This finding indicates that developers do not seem to pay attention to comments in their code during refactoring tasks.

- **STMTC:** We observe from the results in Table 5 that STMTC is impacted by only *Move Attribute* which reduce significantly the STMTC metric by 5, but with a negligible effect size.

- **NIM:** As shown in Table 5, only *Move Method* improves the NIM metric by a median value of 1, with a negligible effect size. However, we notice also that two refactorings caused the metric to deteriorate including *Move Attribute* and *Move Class* by a median value of 1, each with a negligible effect size.

- **NIV:** The results of the NIV metric depicted in Table 5 indicate that only *Extract Super Class*, *Extract Sub Class* refactorings improve the NIV metric by decreasing the median by a value ranging from 1 to 2 with a negligible effect size.

- **VQTY:** We notice from Table 5 that *Inline Method* is the most influential

refactoring which improve the metric by decreasing the median value by 8 with a medium effect size. Moreover, various other refactorings improve the VQTY, but with less impact by a median value ranging between 1 and 3 including *Extract Interface*, *Extract Class*, *Pull Up Method*, *Pull Up Attribute*, *Extract Super Class* and *Extract Sub Class* with a negligible or small effect size.

It is worth noting that our results match also with desktop applications [6, 58]. As an illustrative example, we refer to the LucidLink[8] app, Commit #3b68da9 [64] which implements a *Move And Inline Method*. The refactoring moves the method `access1()` from the class `BluetoothSetup` to the class `AlarmReceiver` where the method is inlined to the `onReceive(context Context, intent Intent)` method which clearly resulted in a reduced size with a drop of its LOC from 66 to 50, its STMTC from from 23 to 20, its NIM from 11 to 9, its NIV from 19 to 15 and its VQTY from 29 to 22.

Finding 1.6. *Most refactoring types tend to reduce the design size metrics except BLOC and CLOC which indicates that developers do not seem to pay attention to comments in the code during their refactoring tasks. The most influential refactoring is Inline Method with a medium effect size.*

Looking at the control group results from Table 5, we noticed that the different quality metrics did not exhibit any significant change with non-refactoring changes (control group), except for the LOC metric that tend to increase after each commit. Indeed, this result is inline with Lehman's law on software evolution [65] indicating that it is typical that the size of a software system increases over time as the project evolves. Hence, our DiD-based statistical model provides evidence that the metrics changes observed in the experiment data are due to refactoring activities and not to chance.

Table 6 shows the lists of corresponding refactorings for each quality attribute in this work where we reduced the noise by considering only commits

[8] `https://github.com/Venryx/LucidLink/`

that contain one refactoring and the previous one [16]. Refactoring names listed in bold means that they are applied only in one of the studies. For instance, we can observe that *Extract Method* improves coupling in the previous study but this is not the case for the current results. Looking at Table 6 we conclude that:

- For cohesion, the results obtained by both studies differ since none of the impactful refactorings identified in the previous study are different from those identified in the current study.
- *Extract Method* and *Push Down Method* yielded an improvement in coupling only in the previous study. However, our results show that several other refactorings including *Extract Interface*, *Move And Inline Method*, *Move And Rename Method* and *Push Down Method* improve coupling. It is worth noting that these refactorings were not considered in the previous study.
- *Push Down Method* is the only common refactoring that improve the inheritance in the two studies.
- All the joint refactorings in the two studies, except *Extract Method* that were identified only in the previous study, improve the design size. However, we observe that the majority of the additional refactorings that are considered in this study improve the design size.

In conclusion, we notice that there are several similarities when comparing the two studies, except for the cohesion attribute where the identified refactorings are totally different. In addition, we noticed that several refactoring types that are added in this study improve the considered quality attributes.

4.2. RQ2. What are the most common co-occurrent refactorings applied by Android developers?

Our motivation from studying the phenomenon of co-occurrent refactorings stems from the observation there is a large number of commits that contain multiple refactoring operations. Indeed, we observe from Table 3 that over

Table 6: Comparison of the refactorings applied in this study versus the previous one [16].

Applied refactorings		
Study	Previous study [16]	Current Study
Cohesion	**Move Attribute, Move Method, Extract And Move Method**	**Pull Up Method, Extract Super Class, Extract Sub Class, Push Down Attribute**
Coupling	**Extract Method**, Move Method, Move Attribute, Extract And Move Method, Inline Method, **Push Down Method**, Pull Up Method, Extract Super Class, Move Class	Move Method, Move Attribute, Inline Method, Pull Up Method, **Extract Class**, Extract Super Class, **Extract Interface**, Move Class, **Move And Inline Method**, **Push Down Attribute**
Complexity	Extract Method, Move Method, Extract And Move Method, **Pull Up Attribute**, Pull Up Method, Extract Super Class, Move Class	Extract Method, Move Method, Move Attribute, **Inline Method**, Pull Up Method, **Extract Class**, **Push Down Method**, Extract Super Class, **Extract Interface**, Move Class, **Push Down Attribute**
Inheritance	**Pull Up Method**, Push Down Method, **Pull Up Attribute**, **Extract Super Class**, **Move Class**	Push Down Method,**Push Down Attribute**
Design size	**Extract Method**, Move Attribute, Move Method, Extract And Move Method, Push Down Method, Pull Up Method, Pull Up Attribute, Extract Super Class, Move Class	Move Method, Move Attribute, **Inline Method**, Pull Up Method, Pull Up Attribute, **Extract Class**, Push Down Method, Extract Super Class, **Extract Sub Class**, **Extract Interface**, Move Method, **Push Down Attribute**

60% (22,009 out of 36,681) of refactoring commits contain multiple refactoring types. Hence, we investigate these commits to discover patterns of co-occurrent refactorings that may occur in the studied applications. We use the Apriori algorithm to determine the associations between the different applied refactoring operations in the same commits. To generate frequent itemsets, we selected a minimum confidence of 0.5. Table 7 presents the frequent itemsets where each itemset comprises two refactoring types. It is worth noting that we did not found any itemset composed by more than two refactoring types at this confidence level. We also conduct Chi-squared and Cramer's V tests to check whether the associations between refactorings are statistically significant or not. It is also worth noting that we found some reciprocal associations with some variations in the confidence value. As can be observed from the table, we mainly found that some refactoring operations tend to co-occur frequently with other refactorings leading to 16 co-occurrent refactorings. In the following we describe these co-occurrent refactorings.

- *Extract Method* refactoring is often associated with various other code refactoring types, and in particular with *Extract Interface*, *Rename Method* and *Move Method*. For the association with the *Rename Method*, this is an

intended outcome since the method undergoing an extract method refactoring is generally also renamed to reflect its new purpose. As for the association with the *Move Method* refactoring, we found after performing some manual analysis that in many cases Android developers tend to extract and move methods for many reasons depending on their intention (*e.g.,* improving code structure). Finally, the strong association with the *Extract Interface* was not intuitive. Hence, we performed a manual analysis to understand the reasons. We found that each of them is applied for a specific reason such as extracting a part of a method to be moved to the extracted interface which also justifies the co-occurence with *Move Method* and *Rename Method*.

- *Extract Interface* refactoring is often co-applied with three refactoring types, namely *Extract Method*, *Move Method* and *Move Attribute*. The *Extract Interface* and *Move Method/Attribute* refactorings do co-occur since when developers extract an interface, they are moving the identical portion of the main interface to its own one, and thus they will apply moving related refactorings.

- *Inline Method* often co-occurs with *Move Method* refactoring. We conduct a qualitative investigation on various samples of co-occurrent refactorings, and we simply found that generally developers inline and move methods for various reasons (*e.g.,* improving code structure, simplifying code, etc.).

- *Move Method* tend to co-occur with 5 other refactoring types, namely *Extract Interface*, *Extract Method*, *Inline Method*, *Move Attribute* and *Push Down Attribute*. For the association with *Move Attribute* and *Push Down Attribute* refactorings the result is likely to be expected since when developers move a method from its own class to anther class, they should certainly move the attributes used by this methods. As for the association with *Extract Method* and *Inline Method* their application depends on the intention of developer (*e.g.,* the intention of increasing class cohesion or coupling, etc.).

- *Push Down Attribute* often co-occurs with *Move Attribute*. It is worth noting that this association is reciprocal as shown in Table 7. This could be an expected consequence since their strong association with the *Move Method* refactoring increases their chance to be associated.

To illustrate some of these co-occurrence patterns, we report an example of co-occurrent refactorings from a particular commit in the IRremote[9] application [66]. In particular, the `updateRemotesList()` method was extracted from the `init()` method in the class `SelectRemoteListView` and was moved to the `MyAdapter` class. Also, the `getView` method was moved from the `SelectRemoteListView` to the `MyAdapter` class. Thus, the developer applied an *Extract And Move Method* refactoring followed by a *Move Method* refactoring in the `SelectRemoteListView` class. Such refactorings are typically inter-related, and therefore tend to co-occur frequently in the same commit as Android developers endeavor to clean up their code.

> ***Finding 2.*** *Co-occurrent refactorings are quite prevalent in Android applications. Overall, 60% of the total number of refactoring commits contain multiple refactoring types, and several co-occurrent refactorings (16) tend to occur very often (e.g., Extract Method and Rename Method).*

4.3. RQ3. How do co-occurrent refactorings affect quality metrics?

In this section, we assess the impact of co-occurrent refactorings identified in RQ2 on different internal quality metrics. To do so, we first identify the refactoring pairs identified in RQ2 regardless their order found by the association rule mining (for example, the two refactoring pairs "`Move Attribute : Move Method`" and "`Move Method : Move Attribute`" are considered as a unique co-occurent refactoring) leading to 10 unique co-occurrent refactorings. There-

[9] `https://github.com/Arduino-IRremote/Arduino-IRremote`

Table 7: The most common co-occurrent refactorings

Refactoring item set #1	Refactoring item set #2	Support	Confidence	Lift	Chi-square p-values	Cramer's V
Extract Method	Rename Method	0.091	0.967	5.948	<0,05	0.487
Extract Method	Move Method	0.074	0.900	3.471	<0,01	0.393
Extract Method	Extract Interface	0.071	0.624	1.153	<0,01	0.488
Extract Interface	Extract Method	0.071	0.757	1.153	<0,01	0.488
Extract Interface	Move Attribute	0.073	0.546	2.278	<0,01	0.620
Extract Interface	Move Method	0.061	0.595	1.575	<0,01	0.440
Inline Method	Move Method	0.083	0.991	1.524	<0,01	0.389
Move Method	Extract Interface	0.061	0.790	1.575	<0,01	0.440
Move Method	Extract Method	0.074	0.868	2.471	<0,01	0.393
Move Method	Inline Method	0.083	0.667	1.524	<0,01	0.389
Move Method	Move Attribute	0.075	0.917	1.364	<0,01	0.429
Move Method	Push Down Attribute	0.063	0.565	1.572	<0,05	0.504
Move Attribute	Move Method	0.075	0.875	1.364	<0,01	0.429
Move Attribute	Extract Interface	0.073	0.544	2.278	<0,01	0.620
Push Down Attribute	Move Attribute	0.035	0.772	1.211	<0,05	0.500
Pull Up Method	Move Attribute	0.084	0.548	6.203	<0,05	0.538

after, we compute for each co-occurent refactoring, its corresponding metric values before and after each commit.

Table 8 show the metrics values before and after applying each co-occurrent refactorings. Overall, the table reports 210 measurements on the experimental group, *i.e.,* 10 co-occurent refactorings × 21 quality metrics. The results indicate that in 54.3% of the cases (114 out of 210 measurements as represented in the while color in the table), refactoring have no statistically significant impact on quality metrics, while it has a positive impact in 42.4% of the cases (89 out of 210 as presented in blue) and a negative impact in 3.3% of the cases (7 out of 210 as presented in red). In the following, we report and discuss the obtained results for each quality metric along with real world examples from our experiments.

Finding 3.1. *Co-occurrent refactorings have no statistically significant impact on quality metrics in 54.3% of the cases, a significant positive impact in 42.4% of the cases, and a significant negative impact in 3.3% of the cases.*

4.3.1. Results for Coupling

- **CBO:** From Table 8, we observe that the co-occurrent refactoring *(Move Method : Push Down Attribute)* is the most influential one which decreases

the median value by 8 accompanied with a negligible effect size. Whereas, *(Move Method : Move Attribute)* and *(Extract Method : Rename Method)* co-occurrences tend to have less impact on the metrics with a median improvement of 3 and 1, respectively, with a negligible effect size, each. These results are in-line with the findings in RQ1 since the applied co-occurrences contain either a *Move Method* or *Extract Method* refactoring. The method-level move refactorings help organizing functionalities across classes, and thus reduce dependencies between them which results in a reduced CBO. As for the *(Extract Method : Rename Method)* co-occurrence, extracting a code fragment from methods can decrease import coupling as it removes dependencies which are duplicated across these methods.

- **RFC:** As can be seen from Table 8, several co-occurrent refactorings do impact the RFC metric. We observe that *(Extract Method : Extract Interface)*, *(Extract Interface : Move Method)*, *(Extract Interface : Move Attribute)*, *(Extract Method : Move Method)* and *(Extract Method: Inline Method)* are the most influential co-occurrences that improve the RFC by decreasing the median ranging from 11 to 12 accompanied with a medium or small effect size depending on the co-occurrent refactoring. Moreover, *(Move Method : Push Down Attribute)*, *(Move Attribute : Move And Inline Method)* and *(Move Attribute : Pull Up Method)* have shown to improve RFC by a median ranging from 7 to 10 with a medium or small effect size, while less impact is observed by *(Move Method : Move Attribute)* and *(Extract Method : Rename Method)* co-occurrences with a median score ranging from 2 to 3 with a small or negligible effect size.

- **FANIN, FANOUT and NOSI:** From Table 8, we observe that the FANIN, FANOUT, NOSI metrics did not experienced an impact with any of the applied co-occurrent refactorings since the DiD model did not show any statistically significant differences.

One of the examples that shows an improvement in coupling was found in the

GymDiary application [10], in Commit #cb5a04c [67]. Specifically, the developer applied the *(Move Method : Move Attribute)* co-occurrent refactoring involving the `TrainingAtProgress` and `BasicMenuActivityNew` classes. This was realized through moving the `onChoose()` method as well as the `adapter` and `sp` attributes from the `TrainingAtProgress` class to the `BasicMenuActivityNew`, resulting in its CBO dropping from 27 to 24 and its RFC from 34 to 29.

Finding 3.2. *Co-occurrent refactorings have a significant positive impact on coupling in terms of both CBO and RFC metrics, while no significant impact was found on the FANIN, FANOUT and NOSI metrics. The co-occurrences that most influence cohesion contain either moving or extracting related refactorings (e.g., Move Method : Push Down Attribute).*

4.3.2. Results for Cohesion

- **LCOM:** Table 8 shows that LCOM is improved when applying the *(Extract Method : Rename Method)* co-occurrent refactoring. The median value significantly decreased by 2, even though it is accompanied by a small effect size.

- **TCC:** We observe from Table 8 that *(Extract Method : Extract Interface)* and *(Move Method : Move Attribute)* are the only two co-occurrent refactorings that improve TCC by increasing the median value by 1 with a negligible effect size, each.

An interesting example that shows the impact of the *(Move Method : Move Attribute)* co-occurrent refactoring on cohesion was found in the Harris Cam[11] application from Commit #2055774 [68] that involves moving the methods `showOptionMenu()` and `setOnClickOptionMenu()` from the `SlideMenuView` class to `RightSlideMenuView` and `setOnClickOptionMenu` classes, respectively. Moreover, nine attributes used by the old methods (*e.g.,* `llSubContainer`,

[10] `https://github.com/nethergrim/GymDiary`
[11] `https://github.com/datakun/HarrisCam/`

Table 8: The impact of co-occurrent refactorings on quality metrics.

Co-occurrent refactorings	Measure	Cohesion		Coupling					Complexity				Inheritance			Design Size						
		LCOM	TCC	CBO	RFC	FANIN	FANOUT	NOSI	WMC	EVG	MaxNest	CC	DIT	NOC	IFANIN	LOC	BLOC	CLOC	STMTC	NIM	NIV	VQTY
Extract Method : Extract Interface	Refactoring Impact	0	1	0	-12	0	0	0	-5	-3	-3	-1	0	0	0	-2	0	0	7	2	-20	0
	Behavior	-	↑	-	↓	-	-	-	↓	↓	↓	↓	-	-	-	↓	-	-	↑	↑	↓	-
	P-value (δ)	1 (N)	<0.05 (S)	1 (N)	<0.05 (M)	0.07 (N)	0.07 (N)	<0.05 (N)	<0.05 (S)	<0.05 (S)	<0.05 (S)	<0.05 (N)	0.08 (N)	0.23 (S)	0.19 (N)	0.06 (N)	0.16 (N)	<0.05 (N)	0.09 (S)	<0.05 (N)	<0.05 (M)	0.12 (N)
Extract Method : Rename Method	Refactoring Impact	-2	0	-1	-3	0	0	0	-1	-2	-2	-1	-1	0	0	-6	0	0	3	1	5	2
	Behavior	↓	-	↓	↓	-	-	-	↓	↓	↓	↓	↓	-	-	↓	-	-	↑	↑	↑	↑
	P-value (δ)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (S)	0.23 (N)	0.09 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (S)	<0.05 (N)	0.65 (N)	0.16 (N)	0.09 (N)	<0.05 (N)	0.29 (N)	0.07 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)
Extract Method : Move Method	Refactoring Impact	0	0	0	-11	0	0	0	-1	-1	-2	-1	-2	0	0	-2	-4	0	-1	0	-16	-4
	Behavior	-	-	-	↓	-	-	-	↓	↓	↓	↓	↓	-	-	↓	↓	-	↓	-	↓	↓
	P-value (δ)	0.95 (N)	<0.05 (N)	0.28 (N)	<0.05 (M)	0.07 (N)	0.92 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	0.26 (N)	<0.05 (N)	<0.05 (S)	0.38 (N)	<0.05 (N)	0.43 (N)	<0.05 (M)	<0.05 (S)
Extract Interface : Move Method	Refactoring Impact	0	0	0	-12	0	0	0	0	0	-2	-1	-2	0	0	-7	-4	0	-1	0	-8	-2
	Behavior	-	-	-	↓	-	-	-	-	-	↓	↓	↓	-	-	↓	↓	-	↓	-	↓	↓
	P-value (δ)	0.85	0.06 (N)	0.78 (N)	<0.05 (S)	1 (N)	0.09 (N)	0.15 (N)	0.58 (N)	0.06 (N)	<0.05 (S)	<0.05 (N)	<0.05 (N)	0.06 (N)	1 (N)	<0.05 (N)	<0.05 (N)	0.26 (N)	<0.05 (N)	0.85	<0.05 (S)	<0.05 (S)
Extract Interface : Move Attribute	Refactoring Impact	0	0	0	-12	0	0	0	0	0	-2	-1	-2	0	0	-19	-4	0	-1	-1	-1	-1
	Behavior	-	-	-	↓	-	-	-	-	-	↓	↓	↓	-	-	↓	↓	-	↓	↓	↓	↓
	P-value (δ)	0.41 (N)	0.10 (N)	0.78 (N)	<0.05 (M)	0.27 (N)	0.07 (N)	0.09 (N)	0.58 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	0.12 (N)	<0.05 (N)	<0.05 (N)	<0.05 (M)	<0.05 (S)	0.13 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	0.16 (N)
Move Method : Inline Method	Refactoring Impact	0	0	0	-11	0	0	0	-1	0	-2	-1	-2	0	0	-7	-6	0	-1	0	-1	-5
	Behavior	-	-	-	↓	-	-	-	↓	-	↓	↓	↓	-	-	↓	↓	-	↓	-	↓	↓
	P-value (δ)	0.39 (N)	0.06 (N)	0.27 (N)	<0.05 (M)	0.19 (N)	0.60 (N)	0.17 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	0.77 (N)	<0.05 (S)	<0.05 (S)	0.10 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)
Move Method : Move Attribute	Refactoring Impact	0	1	-3	-2	0	0	0	0	0	-1	-5	0	0	0	-3	0	0	-1	-1	0	-1
	Behavior	-	↑	↓	↓	-	-	-	-	-	↓	↓	-	-	-	↓	-	-	↓	↓	-	↓
	P-value (δ)	0.50 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	0.48 (N)	0.08 (N)	0.39 (N)	0.11 (N)	<0.05 (N)	<0.05 (N)	<0.05 (S)	1 (N)	0.9 (N)	0.74 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	0.19 (N)	<0.05 (N)
Move Method : Push Down Attribute	Refactoring Impact	3	0	-8	-10	0	0	0	-6	-1	-1	-1	-6	1	-1	-25	6	-3	-3	0	1	-6
	Behavior	↑	-	↓	↓	-	-	-	↓	↓	↓	↓	↓	↑	↓	↓	↑	↓	↓	-	↑	↓
	P-value (δ)	1 (N)	0.12 (N)	<0.05 (N)	<0.05 (S)	0.33 (N)	0.17 (N)	<0.05 (N)	<0.05 (S)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	1 (N)	1 (N)	<0.05 (M)	<0.05 (M)	0.9 (N)	<0.05 (S)	0.9 (N)	1 (N)	<0.05 (S)
Move Attribute : Push Down Attribute	Refactoring Impact	0	0	0	-1	0	0	0	0	0	-2	-1	0	0	0	-8	-4	0	-1	-1	-2	0
	Behavior	-	-	-	↓	-	-	-	-	-	↓	↓	-	-	-	↓	↓	-	↓	↓	↓	-
	P-value (δ)	1 (N)	1 (N)	1 (N)	0.15 (N)	0.11 (N)	0.69 (N)	1 (N)	1 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	1 (N)	<0.05 (N)	1 (N)	<0.05 (S)	<0.05 (S)	1 (N)	<0.05 (N)	0.9 (N)	<0.05 (N)	0.78 (N)
Move Attribute : Pull Up Method	Refactoring Impact	0	0	0	-7	0	0	0	0	-1	-3	-1	5	0	0	-19	-12	0	-2	-2	-2	-5
	Behavior	-	-	-	↓	-	-	-	-	↓	↓	↓	↑	-	-	↓	↓	-	↓	↓	↓	↓
	P-value (δ)	0.55 (N)	1 (N)	0.30 (N)	<0.05 (M)	0.21 (N)	0.79 (N)	0.59 (N)	0.15 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	0.06 (N)	0.34 (N)	<0.05 (M)	<0.05 (M)	0.19 (N)	<0.05 (N)	<0.05 (N)	<0.05 (N)	<0.05 (S)

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.

`ibFlashlight`, `ibGuideline`, `ibCameraSwitcher` and `ibIntervalWatch`) are moved to the new class. These co-occurrent refactorings improve cohesion by decreasing the LCOM metric from 78 to 73 as well as decreasing the TCC metric by 0.8.

Finding 3.3. *Three co-occurrent refactorings tend to generally improve the cohesion quality attribute in terms of the LCOM and TCC metrics including "Move Method : Move Attribute", "Extract Method : Extract Interface" and "Extract Method : Rename Method".*

4.3.3. Results for Complexity

- **WMC:** We observe from Table 8 that *(Move Method : Push Down Attribute)* and *(Extract Method : Extract Interface)* are the most two influential co-occurrent refactorings that improve the WMC with a median value by 6 and 5, respectively with a small effect size, each. Moreover, *(Extract Method : Rename Method)*, *(Extract Method : Move Method)* and *(Move Method : Inline Method)*, but with less impact with a median value of 1 and a negligible effect size, each.

- **EVG:** As can be seen from Table 8, *(Extract Method : Extract Interface)* and *(Extract Method : Rename Method)* are the most influential co-occurrent refactorings that improve the median value by 3 and 2, respectively accompanied with a small effect size. Furthermore, *(Extract Method : Move Method)*, *(Move Method : Push Down Attribute)*, and *(Move Attribute : Pull Up Method)*, but with a median value of 1 and negligible effect size, each.

- **MaxNest:** We notice from Table 8 that all the applied co-occurrent refactorings do improve the MaxNest metric but with a different impact level where the median values ranging from 1 to 2 with a small or negligible effect size.

- **CC:** As can be seen from Table 8, all the applied co-occurrent refactorings do improve the CC metric. *(Move Method : Move Attribute)* is the most

influential co-occurrence that improves the metric with a median value of 5 with a small effect size and the rest of co-occurrences improve the metric with a median value of 1 and a negligible effect size, each.

As an illustrative example, we refer to the NAO-Com[12] app, Commit # 4406253 [69] which implements an *(Extract Interface : Extract Method)* co-occurrent refactoring. The `NetworkDataSender` interface is extracted from the `NAOConnector` class and the method `disconnect()` is extracted from the method `run()` in the `NAOConnector` class which clearly reduced the overall class complexity complexity by decreasing the WMC from 38 to 30, EVG from 2 to 1, MaxNest from 8 to 4 and CC from 3 to 1.

Finding 3.4. *Several co-occurrent refactorings tend to improve complexity by decreasing the WMC, EVG, MaxNest and CC metrics. The most impactful co-occurrent refactorings are "Extract Method : Extract Interface" which typically help simplifying methods structure and/or reducing duplicate code.*

4.3.4. Results for Inheritance

- **DIT:** We notice from Table 8 that five from all the applied co-occurrent refactorings do improve the DIT metric, including *(Move Method : Push Down Attribute)*, *(Move Method : Inline Method)*, *(Extract Interface : Move Method)*, and *(Extract Method : Move Method)*. However, we also observe from the results in Table 8 that the *(Move Attribute : Pull Up Method)* co-occurrence caused DIT metric to disprove by a median of 5 with a negligible effect size.

- **NOC and IFANIN:** From Table 8, we observe that both NOC and IFANIN metrics have not been impacted by any of the applied co-occurrent refactorings since the DiD model did not show any statistically significant differences.

[12] `https://github.com/NorthernStars/NAO-Com`

One of the examples that shows an improvement in inheritance was found in a particular commit in the HIITMe application [70]. Specifically, the developer applied a *Move Method and Push Down Attribute* co-occurrence in the `DraggableView` class. In particular, the `initialise` method as well as the `dragManager` and *ProgramNodeView* attributes were moved from the `DraggableView` class to the `ExerciseView` class. These changes resulted in inheritance improvement for the `DraggableView` class, with a drop of its DIT from an overly-complex 9 to a more manageable 6.

Finding 3.5. *Hierarchy-level and moving related co-occurrent refactorings tend to improve the inheritance quality attribute in terms of DIT, while no significant impact was found on the NOC and IFANIN metrics. The co-occurrent refactorings that most influence inheritance are "Move Method : Push Down Attribute" and "Move Attribute : Move And Inline Method".*

4.3.5. Results for Design Size

- **LOC:** As shown in Table 8, all the applied co-occurrences improve the LOC metric. The most influential co-occurrence is *(Move Method : Push Down Attribute)* which decrease the median value by 25 with a medium effect size. Moreover, *(Extract Interface : Move Attribute)*, *(Move Attribute : Move And Inline Method)* and *(Move Attribute : Pull Up Method)* have shown to improve LOC by a median score of 19 with a medium effect size, while less impact is observed by the rest of co-occurrences.

- **BLOC:** We observe from Table 8 that seven out of all the co-occurrences improve the BLOC metric. The most influential ones are *(Move Attribute : Move And Inline Method)* and *(Move Attribute : Pull Up Method)* which decrease the median value by 13 and 12, respectively with a medium effect size, each. Furthermore, *(Move Method: Inline Method)*, *(Extract Method: Move Method)*, *(Extract Interface : Move Method)*, *(Extract Interface : Move Attribute)* and *(Move Attribute : Push Down Attribute)* tend to significantly improve BLOC by a median value ranging from 4

to 6, each with a small or negligible effect size. However, we also notice from the results in Table 8 that *(Move Method : Push Down Attribute)* co-occurrence caused BLOC metric to disprove by a median of 6 with a medium effect size.

- **CLOC:** As can be seen from Table 8 also CLOC is not impacted by any of the applied co-occurrences. This result indicates that developers do not seem to pay attention to comments in the code during their refactoring tasks.

- **STMTC:** As shown in Table 8, 9 out of all co-occurrences improve the metric with a median value ranging from 1 to 3. However, *(Extract Method : Rename Method)* co-occurrence deteriorate the metric by increasing the median value of 3 accompanied with a negligible effect size.

- **NIM:** We observe from Table 8 that four from all the applied co-occurrences including *(Move Attribute : Pull Up Method)*, *(Move Attribute : Move And Inline Method)*, *(Move Method : Move Attribute)* and *(Extract Interface: Move Attribute)* do improve the NIM metric with a median value ranging from 1 to 2 with a negligible effect size. However, we notice that *(Extract Method :Extract Interface)* and *(Extract Method : Rename Method)* co-occurrences disprove the metric by increasing the median by 2 and 1, respectively with negligible effect size.

- **NIV:** From Table 8, we observe that *(Extract Method : Extract Interface)* and *(Extract Move : Move Method)* are the most influential co-occurrences which decrease the median value by 20 and 16, respectively with a medium effect size. Moreover, *(Extract Interface : Move Method)* tend also to improve NIV with a medium impact of 8 with a small effect size. Whereas, the rest of co-occurrent refactorings tend to have less impact on the metrics with a median value ranging between 1 and 2 with a negligible effect size. However, we notice that *(Extract Method : Rename Method)* co-occurrence disprove the metric by increasing the median by 5 with negligible effect

size.

- **VQTY:** As indicated in Table 8, the co-occurrent refactorings *(Move Method : Push Down Attribute)*, *(Move Attribute : Pull Up Method)* and *(Move Method : Inline Method)* are the most influential that improve the VQTY with a median ranging from 5 to 6 accompanied with a small effect size.

As an illustrative example, we refer to the ShaderEditor[13] app, Commit #bb324ba [71] which implements a *(Move Method : Inline Method)* co-occurrent refactoring. The developer moves the *newDropDownView()* method from the class `ShaderAdapter` to the class `ShaderSpinnerAdapter` and then inlines the method `shaderMenuItemSelected()` with the method `onOptionsItemSelected()`. These co-occurrent refactorings clearly resulted in a reduced method size with a drop of its LOC from 80 to 72, its STMTC from from 31 to 29, its NIM from 18 to 16, its NIV from 25 to 19 and its VQTY from 42 to 35.

Finding 3.6. *Most co-occurrent refactorings tend to reduce the design size metrics except for the CLOC which indicates that developers do not seem to pay attention to comments in the code during their refactoring tasks. The most influential co-occurrent refactorings are "Extract Interface : Move Method", "Move Method : Inline Method" and "Move Attribute : Pull Up Method" which typically help reducing duplicate code, or moving code between classes, hence improving the design size.*

In Table 9, we compare the impact of single refactorings (findings of RQ1) against co-occurrent refactorings (findings of RQ3) in improving or worsening the quality attributes. To do so, for each attribute, we identified the metrics that are impacted by at least one single or co-occurrent refactorings.

In summary, we observed some consistency of results obtained for both single and co-occurrent refactorings while more metrics tend to deteriorate with

[13] `https://github.com/markusfisch/ShaderEditor`

co-occurrent refactorings. Such results contrast with the assumption that the metrics changes are due to chance and confirms that regardless of the employed refactoring strategy (*i.e.,* single or co-occurrent refactorings), refactorings generally tend to have an impact on these metrics.

Table 9: Comparison of the results found for single refactorings (RQ1) versus co-occurrent refactorings (RQ3).

	Impacted metrics	
Quality attribute	Single refactorings (RQ1)	Co-occurrent refactorings (RQ3)
Cohesion	LCOM	LCOM, TCC
Coupling	CBO, RFC, FANIN, FANOUT	CBO, RFC
Complexity	All the metrics	All the metrics
Inheritance	DIT, NOC, IFANIN	DIT
Design size	LOC, STMTC, NIM, NIV, VQTY	LOC, BLOC, STMTC, NIV, NIM, VQTY

5. Implications and Discussions

5.1. 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 [72] showed 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 knowl-

edge is needed as, in practice, the built model should require as little data as possible.

Provide knowledge based on current Android applications development practices. There is a growing interest on the detection [73–76], prioritization [75, 77–79], and recommendation [15, 73, 80–82] of refactoring in both academia and industry. Researchers are encouraged to explore such interests together with the practice of refactoring. Several new research opportunities can be explored by devising refactoring tools dedicated for Android applications to incorporate co-occurrent refactorings instead of only focusing on individual refactoring. This is important in the context of mobile applications since such applications need to evolve quickly to meet users and market needs. Indeed, our study showed that over 60% of refactoring commits contain two or more refactorings which reveals that co-occurrent refactorings are common practice. Furthermore, these refactorings significantly improve quality metrics. Moreover, researchers should look into the reasons why developers apply only specific refactoring types and evaluate their perceptions (*i.e.,* why they apply refactoring) in the context of mobile applications.

Control the impact of refactoring. Although, our results showed that a limited number of metrics deteriorate after applying single and/or co-occurrent refactorings, this study identified a research opportunity to control the impact of refactoring on mobile applications evolution. Thus, when performing refactoring, it is also necessary to assess its impacts. For instance, based on our results, researchers could evaluate most commonly used refactorings by mobile applications developers that affect negatively the quality and investigate the reasons behind it while accommodating developers with appropriate quality-aware refactoring recommendation tools especially for mobile applications.

Further investigate co-occurrent refactorings. Previous empirical studies reported that Extract Method and Rename Method are the most common refactoring types applied by developers [83]. These studies may give an intuition that developers tend to most commonly apply single refactorings. However, this is not the case since our results show that developers tend to most commonly

apply co-occurrent refactorings where 60% of commits contain multiple refactoring and 16 refactoring pairs tend to co-occur together (*e.g.,* Extract Method : Rename Method). However, it is still unclear if these co-occurrent refactorings are associated to the same task (*e.g.,* decrease code complexity). Therefore, there is a need to further investigate the intention of Android developers when they apply co-occurrent refactorings.

Further investigate refactoring impact and developer intention behind refactoring application. Our findings showed that some quality metrics deteriorate after the application of certain single and co-occurring refactorings. However, we cannot conclude from this that developers are misapplying refactorings for two reasons. Firstly because it is known that software metrics frequently disagree on whether the application of a refactoring is beneficial or not [11], so the observed deterioration may be due to weaknesses in the metrics themselves — certainly software metrics cannot be taken as ground truth measures of actual quality. Secondly, because we do not know the intention of the developer when they apply a refactoring. Indeed, several recent works have shown that developers tend not to explicitly mention their intention when documenting refactoring related code changes, including refactoring documentation in commit messages [10, 42, 84–86], refactoring documentation in code review [87, 88], as well as refactoring documentation in issue handling [89]. Therefore, as future work it is important to further investigate why some quality metrics deteriorate after refactoring based on the intention of developers when they apply single or co-occurrent refactorings. That is, we speculate that Android developers might be applying sequences of individual or co-occurrent refactoring in different commits (consecutive or not). Looking at the improvement in each commit might not reflect the actual whole improvement; unless combining the improvements together. We encourage future works to explore tracking related refactoring operations over time in different refactoring commits to investigate this phenomenon more deeply.

5.2. Implications for tool builders

Need for tools that recommend co-occurrent refactorings for Android developers. Our results showed 16 common co-occurrent refactorings in practice and their impact on software quality metrics in Android applications. This can be useful for the next generation of refactoring recommendation tools to support co-occurrent multi-purpose refactoring. Moreover, these tools can inform about the side effects and allow the developer to choose the co-occurrent refactorings which may be applied and the side effects to be minimized or removed.

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 applications, it is relevant to recommend refactoring opportunities for classes suffering from specific quality aspects, *e.g.,* coupling, complexity, etc.

Need for tools that predict the effects of refactoring. The main goal of our empirical study is to investigate the impact of refactoring on quality metrics (*i.e.,* decay or improvement). Our results indicate that when refactoring (*i.e.,* single and co-occurrent refactoring) affects the metrics it generally improves them. In many cases refactoring has no significant impact on the metrics, whereas some metrics are disapproved after applying some refactoring types. Therefore, this is an opportunity to propose and/or improve refactoring tools by early predicting and evaluating the effects of refactoring before applying it. Such tools can also allow developers to evaluate different refactoring alternatives based on their quality improvement.

5.3. Implications for practitioners.

Android developers should pay attention to the quality of their application code. Our results indicate that developers sometimes apply refactoring operations that do not improve the structural quality of the application. This is particularly the case for the cohesion metric LCOM. While LCOM tends

to be very volatile under refactoring as also shown in prior works [11, 90], these results indicate that there is a risk that developers degrade application structural quality while performing refactoring changes. Given that Android applications 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 applications. Hence, developers need to pay attention to their refactoring edits.

5.4. 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 applications and in software development, in general. Educators can use our study results and our dataset [19] to teach and motivate students to follow best refactoring practices while avoiding refactoring changes that may cause regression in their applications. In particular, our real world dataset of 84,841 refactorings from 800 Android applications, 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.

6. Threats to validity

This section discusses threats to validity of the study.

Threat to internal validity. The accuracy of the refactoring detection tool, RefactoringMiner, 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 [37, 91], which gives us confidence in using the tool. Furthermore, RefactoringMiner identifies some co-occurrent refactorings such as *Extract and Move Method*, and

Move and Rename Method. We excluded those refactorings from our study as they might bias the final results while focusing on single and co-occurrent refactoring. Hence, further deep investigation of co-occurrent refactoring would be interesting as future work. The set of metrics used in this study also represents a threat to validity, because it may not capture all relevant properties of the internal quality attributes. To mitigate this threat, we did not choose a random set of metrics. We chose metrics that assess different properties of each internal quality attribute [26, 28, 92]. Another criterion was used to choose our set of metrics that are detailed in Section 2.2.

Threats to construct validity. A potential threat to construct validity is related to commit-level changes. Similar to the work of Pantiuchina et al. [56], our study does not exclude tangled commits [93], *i.e.,* commits in which developers perform changes related to different tasks and one of these tasks could be related to refactoring. To reduce the impact of the noise resulting from tangled commits, we employed the Difference-in-Difference (DiD) model where a treatment group (commits with refactoring) and a control group (commits without refactoring) are compared. We did not consider filtering out such changes in this study as tracing back and separating changes in a single commit is an error prone and non-trivial task.

Threats to conclusion validity. Unlike other works on the impact of refactoring on quality metrics [6, 10, 13, 14], we employed the DiD method to compare the changes in quality metrics between a treatment and control group and filtered our dataset to choose. Furthermore, we used the non-parametric Wilcoxon rank-sum test and the Cliff's effect size, that do not make assumptions on the underlying data. Another threat to conclusion validity is concerned with the causal relationship between the treatment and the outcome. The algorithm used to identify co-occurrentrefactorings is important. In fact, different results may be obtained with different other algorithms used in recent studies [23, 94]. Thus, we need to evaluate in our future work the impact of different algorithms on the quality of the results.

Threat to external validity. In this work, we investigate the impact of refac-

toring on quality metrics using a large sample of 800 open source Android applications written in Java. In addition, we have only studied the mobile applications of a single mobile platform (*i.e.,* the Android Platform). Thus, we cannot generalize our results to other open source or commercial mobile applications or to other technologies.

7. Related work

This section reports the literature related to different studies on (*i*) the identification and detection of refactoring activities, and (*ii*) the impact of single and co-occurrentrefactorings refactoring performed by developers on code quality.

7.1. Refactoring: methods and tools

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. [95] 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.

Several researchers have used messages attached to commits into a version control as indicators of refactoring activity. Stroggylos and Spinellis [14] searched words stemming from the verb refactor, such as refactoring or refactored, to identify refactoring-related commits. Ratzinger et al. [96] also used a similar keyword-based approach to detect refactoring activity between a pair of program versions in order to identify whether a transformation contains refac-

toring. The authors identified refactorings based on a set of keywords detected in commit messages (*e.g.,* refactor, restruct, clean, etc).

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.

7.2. Empirical studies on the impact of single refactorings

Various research works attempted to quantitatively evaluate whether refactoring indeed improves quality in traditional software systems. Alshayeb et al. [12] 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. [56] 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. [10] 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. Further, research particularly in resability refactorings [41, 42] examined 1,828 projects and 154,820 commits that modified Java files. The authors considered how reusability changes affect software quality metrics and how what kinds of refactoring operations were performed during reusability changes.

Tahvildari & Kontogiannis [97] 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. [11] 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. [98] 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. [99] 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.

Stroggylos and Spinellis [14] analyzed the version control system logs of several popular open source software systems to determine the impact refactorings have on software metrics, and found that refactoring did not cause the metrics to improve. However this work relied upon the use of commit messages to detect refactorings, an approach that was later discredited [38]. Kataoka et al. [100] 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. [13] 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 10 provides a summary of the existing works.

Recently, Hamdi et al. [16] propose a first analysis on the impact of refactoring on quality metrics in the context of Android applications. They mined 300 open-source applications containing 42,181 refactoring operations in total. They 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

Table 10: A summary of the literature on the impact of refactoring on software quality attributes.

Study	Year	Software Metric	Internal Quality Attribute	Refactoring Level		
				Class	Method	Field
Simon et al. [95]	2001	Cohesion measures	Cohesion	Yes	Yes	Yes
Kataoka et al. [100]	2002	Coupling measures	Coupling	Yes	Yes	No
Tahvildari & Kontogiannis [97]	2004	LCOM / WMC / RFC / NOM CDE / DAC / TCC	Inheritance / Cohesion / Coupling / Complexity	Not mentioned		
Geppert et al. [98]	2005	Not mentioned	Not mentioned	Not mentioned		
Stroggylos & Spinells [14]	2007	CK / Ca / NPM	Inheritance / Cohesion / Coupling / Complexity	Not mentioned		
Alshayeb [12]	2009	CK / LOC / FANOUT	Inheritance / Cohesion / Coupling / Size	Yes	Yes	Yes
Ó Cinnéide et al. [11]	2012	LSCC / TCC / CC / SCOM / LCOM5	Cohesion	Yes	Yes	Yes
Szoke et al. [99]	2014	CC / U / NOA / NII / NAni LOC / NUMPAR / NMni / NA	Size / Complexity	Not mentioned		
Cedrim at al. [13]	2016	LOC / CBO / NOM / CC FANOUT / FANIN	Cohesion / Coupling / Complexity	Yes	Yes	Yes
Pantiuchina et al. [56]	2018	LCOM / CBO / WMC / RFC C3 / B&W / SRead	Cohesion / Coupling / Complexity	Yes	Yes	Yes
AlOmar et al. [10]	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
AlOmar et al. [42]	2020	CK / LOC / CC	Inheritance / Cohesion / Coupling / Complexity / Size	Yes	Yes	Yes
Hamdi et al. [16]	2021	CK / LOC / VQTY	Inheritance / Cohesion / Coupling / Complexity / size	Yes	Yes	Yes
AlOmar et al. [41]	2021	CK / LOC / CC / NPATH MaxNest / FANIN / FANOUT / CLOC STMTC / CDL / NIV / NIM / IFANIN	Inheritance / Cohesion / Coupling / Complexity / Size	Yes	Yes	Yes
This work		CK / FANIN / FANOUT / CC / NIV / NIM Evg / MaxNest / IFANIN / TCC / VQTY LOC / CLOC / BLOC / STMTC / NOSI	Inheritance / Cohesion / Coupling / Complexity Size	Yes	Yes	Yes

the metric changes in non-refactoring commits. The results show that metrics can be a strong indicator for refactoring activity, regardless of whether it improves or degrades these metric values. In particular, 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 the design size metrics.

7.3. Empirical studies on the impact of co-occurrent refactorings

Many studies in traditional software systems have investigate the effect of co-occurrent refactorings. Vinicius Soares et al. [22] present a quantitative study aimed at understanding the most common incomplete co-occurrent refactorings and how they impact internal quality attributes. They selected five software projects of different domains and a set of common refactoring types. Then, they collected incomplete co-occurrent refactorings for Feature Envy removal

or God Class removal and computed the frequency of incomplete co-occurrent refactorings according to the refactoring types constituting each co-occurrent refactoring. Then, they evaluated the effect of those incomplete co-occurrent refactorings on 11 code metrics that are used to capture four internal quality attributes. They found that incomplete co-occurrent refactorings with at least one Extract Method are often (71%) applied without Move Methods on smelly classes. They have also found that most incomplete co-occurrent refactorings (58%) tended to at least maintain the internal structural quality of smelly classes, thereby not causing more harm.

Also, Sousa et al. [23] investigate co-occurrent refactorings. They mined the commit history of 48 GitHub software projects to identify the characteristics of different categories of co-occurrent refactorings, and their impact on either removing or introducing smells. They found that many smells are introduced in a program due to incomplete co-occurrent refactorings. Also, they found that 111 patterns of co-occurrent refactorings frequently introduce or remove certain smell types.

Bibiano et al. [101] present a large-scale quantitative study in 57 open and closed software projects understand the usual co-occurrent application from two perspectives: characteristics that typically constitute a co-occurrent refactoring, and the co-occurrent impact on smells. They analyzed 19 smell types and 13 code refactorings. They found that most co-occurrent refactorings in practice occur entirely within one commit (93%), affect multiple methods (90%) and co-occurrent refactorings mostly end up introducing (51%) or not removing (38%) code smells.

We observe from the existing literature that most studies focus on desktop applications while few works have focused on mobile applications. Furthermore, existing studies are limited to only a small number of quality metrics, or/and few refactoring types. Finally, one of the limitations in the state-of-the-art studies is that they do not partition their dataset to consider single-refactoring commits and multi-refactoring commits separately. Such code changes can add more noise to the analyzed quality metrics values, and impact the final outcome

of the metrics analysis. This paper extends our previous work [16] and is different from other papers cited in this section. In particular, we focus on Android applications, include more quality metrics in our analysis, more refactoring operations and more Android applications. Moreover, we filtered our dataset to consider commits containing only one refactoring. In our study, we adopt a causal inference based on the DiD model [17] to better assess the impact of refactoring on quality metrics and provide greater confidence that the observed metric variation is indeed due to refactoring.

8. Conclusion and Future Work

In this paper we investigated the impact of refactoring on quality metrics in Android applications. To ensure that we have high quality-data we mined 800 open-source applications containing 84,841 a total of refactoring operations. We first analyze the impact of single refactoring operations on 21 common quality metrics using the Difference-in-Difference (DiD) statistical model. Then, we investigated the most common co-occurrent refactorings, and their impact on quality metrics. Our findings reveal that when single refactorings affect the metrics, they generally improve them. However, in many cases refactoring has no significant impact on quality metrics. Moreover, we found that co-occurrent refactorings are quite prevalent in Android applications. Overall, 60% of the total number of refactoring commits contain multiple refactoring types, and 16 co-occurrent refactoring pairs tend to occur very often, where many of them have a positive impact on quality metrics. Our results can have several important implications for researchers, tool builders and educators.

We foresee several possible directions for future work. Firstly, we plan to investigate the impact of co-occurrent refactoring on code smells, in particular to assess the extent to which they are effective at removing smells. Secondly, we consider extending our study to commercial Android applications written with different programming languages to better generalize our results. Finally, we plan to conduct a qualitative investigation through a survey with Android

developers to better understand their intuition behind refactoring activities in the context of Android applications development.

Acknowledgements

This research work has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) RGPIN-2018-05960.

References

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

[2] G. Hecht, N. Moha, 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, 2016, pp. 59–69.

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

[4] A. Uchôa, C. Barbosa, W. Oizumi, P. Blenilio, R. Lima, A. Garcia, 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), IEEE, 2020, pp. 511–522.

[5] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of bad smells in object-oriented code, in: Seventh International Conference on the Quality of Information and Communications Technology, 2010, pp. 106–115.

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

[7] R. Minelli, Software analytics for mobile applications, Roberto Minelli, 2012.

[8] I. J. M. Ruiz, M. Nagappan, B. Adams, A. E. Hassan, Understanding reuse in the android market, in: IEEE International Conference on Program Comprehension (ICPC), 2012, pp. 113–122.

[9] L. Xu, S. F. Wu, H. Chen, Techniques and tools for analyzing and understanding android applications, University of California, Davis, 2013.

[10] E. A. AlOmar, M. W. Mkaouer, A. Ouni, 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), IEEE, 2019, pp. 1–11.

[11] M. Ó Cinnéide, I. Hemati Moghadam, M. Harman, S. Counsell, L. Tratt, An experimental search-based approach to cohesion metric evaluation, Empirical Software Engineering 22 (1) (2017) 292—-329.

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

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

[14] K. Stroggylos, D. Spinellis, Refactoring–does it improve software quality?, in: International Workshop on Software Quality (WoSQ), 2007, pp. 10–10.

[15] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, 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) (2016) 1–53.

[16] O. Hamdi, A. Ouni, E. A. AlOmar, M. O. Cinnéide, M. W. Mkaouer, An empirical study on the impact of refactoring on quality metrics in android

applications, in: 2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft), IEEE, 2021, pp. 28–39.

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

[18] R. Agrawal, T. Imielinski, A. Swami, Mining associations between sets of items in large databases, in: International Conference on Management of Data, 1993, pp. 207–216.

[19] Dataset:, `https://github.com/stilab-ets/AndroidRefactoring` (2022).

[20] W. F. Opdyke, Refactoring: A program restructuring aid in designing object-oriented application frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign (1992).

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

[22] A. C. Bibiano, V. Soares, D. Coutinho, E. Fernandes, J. L. Correia, K. Santos, A. Oliveira, A. Garcia, R. Gheyi, B. Fonseca, et al., How does incomplete composite refactoring affect internal quality attributes?, in: 28th International Conference on Program Comprehension, 2020, pp. 149–159.

[23] L. Sousa, D. Cedrim, A. Garcia, W. Oizumi, A. C. Bibiano, D. Oliveira, M. Kim, A. Oliveira, Characterizing and identifying composite refactorings: Concepts, heuristics and patterns, in: 17th International Conference on Mining Software Repositories, 2020, pp. 186–197.

[24] W. Oizumi, A. C. Bibiano, D. Cedrim, A. Oliveira, L. Sousa, A. Garcia, D. Oliveira, Recommending composite refactorings for smell removal: Heuristics and evaluation, in: 34th Brazilian Symposium on Software Engineering, 2020, pp. 72–81.

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

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

[27] M. Aniche, Ck-metrics, `https://github.com/mauricioaniche/ck`,, accessed: 2020-01-09 (2016).

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

[29] M. Lorenz, J. Kidd, Object-oriented software metrics: a practical guide, Prentice-Hall, Inc., 1994.

[30] G. Destefanis, S. Counsell, G. Concas, R. Tonelli, Agile processes in software engineering and extreme programming. chapter software metrics in agile software: An empirical study (2014).

[31] P. Runeson, M. Host, A. Rainer, B. Regnell, Case study research in software engineering: Guidelines and examples, John Wiley & Sons, 2012.

[32] T. Das, M. Di Penta, 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), IEEE, 2016, pp. 443–447.

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

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

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

[36] Z. Xing, E. Stroulia, Umldiff: an algorithm for object-oriented design differencing, in: IEEE/ACM international Conference on Automated software engineering, 2005, pp. 54–65.

[37] D. Silva, N. Tsantalis, M. T. Valente, Why we refactor? confessions of github contributors, in: ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 858–870.

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

[39] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, 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, 2017, pp. 465–475.

[40] A. Ouni, M. Kessentini, H. Sahraoui, M. S. Hamdi, Search-based refactoring: Towards semantics preservation, in: 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 347–356.

[41] E. A. AlOmar, T. Wang, V. Raut, M. W. Mkaouer, C. Newman, A. Ouni, Refactoring for reuse: An empirical study, Innovations in Systems and Software Engineering.

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

[43] S. Henry, D. Kafura, Software structure metrics based on information flow, IEEE transactions on Software Engineering (5) (1981) 510–518.

[44] B. A. Nejmeh, Npath: a measure of execution path complexity and its applications, Communications of the ACM 31 (2) (1988) 188–200.

[45] G. Destefanis, S. Counsell, G. Concas, R. Tonelli, Agile processes in software engineering and extreme programming, Springer-Verlag, Berlin, Heidelberg, 2014, Ch. Software Metrics in Agile Software: An Empirical Study, pp. 157–170.
URL http://dl.acm.org/citation.cfm?id=2813544.2813555

[46] F. Wilcoxon, S. Katti, 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 (1970) 171–259.

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

[48] M. Kaur, S. Kang, Market basket analysis: Identify the changing trends of market data using association rule mining, Procedia computer science 85 (2016) 78–85.

[49] Z. Jin, Y. Cui, Z. Yan, Survey of intrusion detection methods based on data mining algorithms, in: Proceedings of the 2019 International Conference on Big Data Engineering, 2019, pp. 98–106.

[50] R. Agarwal, Decision making with association rule mining and clustering in supply chains, International Journal of Data and Network Science 1 (1) (2017) 11–18.

[51] S. Brin, R. Motwani, J. D. Ullman, S. Tsur, Dynamic itemset counting and implication rules for market basket data, in: International conference on Management of data, 1997, pp. 255–264.

[52] H. Cramér, Mathematical methods of statistics, Vol. 43, Princeton university press, 1999.

[53] A. C. Bibiano, E. Fernandes, D. Oliveira, A. Garcia, M. Kalinowski, B. Fonseca, R. Oliveira, A. Oliveira, D. Cedrim, A quantitative study on characteristics and effect of batch refactoring on code smells, in: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2019, pp. 1–11.

[54] I. Alazzam, B. Abuata, G. Mhediat, Impact of refactoring on oo metrics: A study on the extract class, extract superclass, encapsulate field and pull up method, International Journal of Machine Learning and Computing 10 (1).

[55] V. E. Zafeiris, S. H. Poulias, N. Diamantidis, E. A. Giakoumakis, Automated refactoring of super-class method invocations to the template method design pattern, Information and Software Technology 82 (2017) 19–35.

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

[57] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, 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, Springer, 2007, pp. 252–266.

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

[59] Move And Inline Method, HIITMe `https://github.com/AlexGilleran/HIITMe/commit/e8b345bf13e4682690fb54ee707f8e131c947220`, (Accessed on 5/12/2021).

[60] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.

[61] Extract Sub Class, GlidePlayer https://github.com/georgevarghese185/GlidePlayer/commit/c8a6d7156942f50826d20d60909705326d9b624e, (Accessed on 5/12/2021).

[62] Extract Class, daily-dozen-androidhttps://github.com/nutritionfactsorg/daily-dozen-android/commit/8aa4b458a41d6ea13864b721abf3941b2e2c163d, (Accessed on 5/12/2021).

[63] Push Down Method, HistoryCleanerProhttps://github.com/JohnNPhillips/HistoryCleanerPro/commit/23852db0342cde4a87fdc570b490aa7f4015caba, (Accessed on 5/12/2021).

[64] Move And Inline Method, HistoryCleanerProhttps://github.com/Venryx/LucidLink/commit/3b68da94b7e516bbd3d46856642573167d921adc, (Accessed on 5/12/2021).

[65] M. M. Lehman, Laws of software evolution revisited, in: European Workshop on Software Process Technology, Springer, 1996, pp. 108–124.

[66] Move Method and Move Method, HistoryCleanerProhttps://github.com/krautwigundrueben/IRRemote/commit/8f57d9a7130db1aac8e8d0422b82a7f29e31e8e4, (Accessed on 5/12/2021).

[67] (Move Method : Move Attribute), GymDiary https://github.com/nethergrim/GymDiary/commit/cb5a04cad25c031afd8b28ca214df301a6014f6b, (Accessed on 05/12/2021).

[68] (Move Method : Move Attribute), Hariss Cam app https://github.com/datakun/HarrisCam/commit/2055774ddd335574fc583039e3fdda1706b30fe6, (Accessed on 22/11/2021).

[69] (Extract Interface : Extract Method), NAO-Com app https://github.com/NorthernStars/NAO-Com/commit/4406253297eef51a0e505597f122fc56ff013a16, (Accessed on 22/11/2021).

[70] Move Method and Push Down Attribute, HI-ITMehttps://github.com/AlexGilleran/HIITMe/commit/e008723aa69a74676ae35a4b1baca30953837608, (Accessed on 5/12/2021).

[71] Move Method and Inline Method, ShaderEditorhttps://github.com/markusfisch/ShaderEditor/commit/bb324bab706b3bd03359b882bd48d392c18acdec, (Accessed on 5/12/2021).

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

[73] I. Verebi, A model-based approach to software refactoring, in: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2015, pp. 606–609.

[74] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, An empirical investigation into the nature of test smells, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 4–15.

[75] A. Ouni, M. Kessentini, S. Bechikh, H. Sahraoui, Prioritizing code-smells correction tasks using chemical reaction optimization, Software Quality Journal 23 (2) (2015) 323–361.

[76] Y. Kataoka, M. D. Ernst, W. G. Griswold, D. Notkin, Automated support for program refactoring using invariants, in: Proceedings IEEE International Conference on Software Maintenance. ICSM 2001, IEEE, 2001, pp. 736–743.

[77] M. Mondal, C. K. Roy, K. A. Schneider, A comparative study on the bug-proneness of different types of code clones, in: 2015 IEEE International conference on software maintenance and evolution (ICSME), IEEE, 2015, pp. 91–100.

[78] V. Alizadeh, M. Kessentini, M. W. Mkaouer, M. Ocinneide, A. Ouni, Y. Cai, An interactive and dynamic search-based approach to software refactoring recommendations, IEEE Transactions on Software Engineering 46 (9) (2018) 932–961.

[79] H. Liu, Q. Liu, Z. Niu, Y. Liu, Dynamic and automatic feedback-based threshold adaptation for code smell detection, IEEE Transactions on Software Engineering 42 (6) (2015) 544–558.

[80] H. Liu, Q. Liu, Y. Liu, Z. Wang, Identifying renaming opportunities by expanding conducted rename refactorings, IEEE Transactions on Software Engineering 41 (9) (2015) 887–900.

[81] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, A. De Lucia, Methodbook: Recommending move method refactorings via relational topic models, IEEE Transactions on Software Engineering 40 (7) (2013) 671–694.

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

[83] M. Paixão, A. Uchôa, A. C. Bibiano, D. Oliveira, A. Garcia, J. Krinke, E. Arvonio, Behind the intents: An in-depth empirical study on software

refactoring in modern code review, in: Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 125–136.

[84] E. A. AlOmar, M. W. Mkaouer, A. Ouni, Toward the automatic classification of self-affirmed refactoring, Journal of Systems and Software 171 (2021) 110821.

[85] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. Newman, A. Ouni, M. Kessentini, How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation, Expert Systems with Applications 167 (2021) 114176.

[86] E. A. AlOmar, J. Liu, K. Addo, M. W. Mkaouer, C. Newman, A. Ouni, Z. Yu, On the documentation of refactoring types, Automated Software Engineering 29 (2022) 1–40.

[87] E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni, M. Kessentini, Refactoring practices in the context of modern code review: An industrial case study at xerox, in: IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2021, pp. 348–357.

[88] E. A. AlOmar, M. Chouchen, M. W. Mkaouer, A. Ouni, Code review practices for refactoring changes: an empirical study on openstack, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 689–701.

[89] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. D. Newman, A. Ouni, An exploratory study on refactoring documentation in issues handling, in: 19th International Conference on Mining Software Repositories (MSR), 2022, pp. 107–111.

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

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

[92] J. M. Bieman, L. M. Ott, Measuring functional cohesion, IEEE transactions on Software Engineering 20 (8) (1994) 644–657.

[93] K. Herzig, S. Just, A. Zeller, The impact of tangled code changes on defect prediction models, Empirical Software Engineering 21 (2016) 303–336.

[94] A. C. Bibiano, W. Assunçao, D. Coutinho, K. Santos, V. Soares, R. Gheyi, A. Garcia, B. Fonseca, M. Ribeiro, D. Oliveira, et al., Look ahead! revealing complete composite refactorings and their smelliness effects.

[95] F. Simon, F. Steinbruckner, C. Lewerentz, Metrics based refactoring, in: Proceedings fifth european conference on software maintenance and reengineering, IEEE, 2001, pp. 30–38.

[96] J. Ratzinger, T. Sigmund, H. C. Gall, On the relation of refactorings and software defect prediction, in: Proceedings of the 2008 international working conference on Mining software repositories, 2008, pp. 35–38.

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

[98] B. Geppert, A. Mockus, F. Robler, Refactoring for changeability: A way to go?, in: 11th IEEE International Software Metrics Symposium, 2005, p. 10.

[99] G. Szóke, G. Antal, C. Nagy, R. Ferenc, 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, IEEE, 2014, pp. 95–104.

[100] Y. Kataoka, T. Imai, H. Andou, T. Fukaya, A quantitative evaluation of maintainability enhancement by refactoring, in: International Conference on Software Maintenance, 2002. Proceedings., IEEE, 2002, pp. 576–585.

[101] A. C. Bibiano, A. Garcia, On the characterization, detection and impact of batch refactoring in practice, in: Anais Estendidos do XI Congresso Brasileiro de Software: Teoria e Prática, SBC, 2020, pp. 165–179.