The Journal of Systems & Software 177 (2021) 110964

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Android code smells: From introduction to refactoring☆

Sarra Habchi [a,*], Naouel Moha [b], Romain Rouvoy [c]

[a] University of Luxembourg, Luxembourg
[b] École de Technologie Supérieure, Canada
[c] University of Lille / Inria / IUF, France

ARTICLE INFO

Article history:
Received 16 September 2020
Received in revised form 31 January 2021
Accepted 26 March 2021
Available online 1 April 2021

ABSTRACT

Object-oriented code smells are well-known concepts in software engineering that refer to bad design and development practices commonly observed in software systems. With the emergence of mobile apps, new classes of code smells have been identified by the research community as mobile-specific code smells. These code smells are presented as symptoms of important performance issues or bottlenecks. Despite the multiple empirical studies about these new code smells, their diffuseness and evolution along change histories remains unclear.

We present in this article a large-scale empirical study that inspects the introduction, evolution, and removal of Android code smells. This study relies on data extracted from 324 apps, a manual analysis of 561 smell-removing commits, and discussions with 25 Android developers. Our findings reveal that the high diffuseness of mobile-specific code smells is not a result of releasing pressure. We also found that the removal of these code smells is generally a side effect of maintenance activities as developers do not refactor smell instances even when they are aware of them.

© 2021 Elsevier Inc. All rights reserved.

1. Introduction

Mobile apps have established themselves as mainstream software systems deployed at scale. Over the last few years, they successfully invaded the software market and retained the interest of end-users. While mobile apps generally rely on the same software development basics as classical software systems, they also manifest some particularities as they run on embedded devices with key performance constraints. This specificity sets the bar high for mobile apps in the sense that they are expected to remain fluid and efficient while continuously performing complex tasks. As a consequence, development practices that do not satisfy these requirements were qualified as *code smells*. In particular, the study of Reimann et al. (2014) proposed a catalogue of Android-specific code smells that violate performance guidelines. These code smells originate from the good and bad practices presented in the official documentation or by developers reporting their experience on blogs. For example, the code smell *No Low Memory Resolver* describes non-compliance with the official recommendation of implementing memory resolvers inside activities.

The research community extended this catalogue and studied different aspects of mobile-specific code smells. The performance impact of these code smells was inspected in multiple studies (Carette et al., 2017; Hecht et al., 2016; Palomba et al., 2019), showing that they can hinder app performance and increase energy consumption. Research works also proposed automated solutions for detecting mobile code smells, like PAPRIKA and ADOCTOR (Hecht et al., 2015a; Palomba et al., 2017). On the empirical side, our previous studies assessed the key role played by developers in the accrual of mobile code smells (Habchi et al., 2018, 2019a) and quantified their survival in the change history. This allowed us to build an initial understanding of what are mobile code smells and how do they hinder app quality. However, as research remains young in this field, we still lack knowledge about various aspects of code smells:

Lack of extent analysis. Studies proposing code smell detection tools quantified code smell instances in Android and iOS apps (Habchi et al., 2017; Hecht et al., 2015a; Palomba et al., 2017). However, none of these studies analysed and compared the diffuseness of code smells of different types. This comparison is important to distinguish the most common code smells and prioritize their detection and refactoring in future studies. Diffuseness analysis is also important to precisely assess code smell prevalence. In particular, a code smell may seem very frequent only because its host entity is very common in source code. Diffuseness analysis alleviates this by measuring precisely how frequent

☆ Editor: Gabriele Bavota.

* Corresponding author.
E-mail addresses: sarra.habchi@uni.lu (S. Habchi), naouel.moha@etsmtl.ca (N. Moha), romain.rouvoy@inria.fr (R. Rouvoy).

https://doi.org/10.1016/j.jss.2021.110964
0164-1212/© 2021 Elsevier Inc. All rights reserved.

is the code smell considering its occurrence chances and its host entity.

Lack of release analysis. Some studies (Habchi et al., 2019a,c) leveraged the change history of mobile apps to better understand code smells. Specifically, our previous work (Habchi et al., 2019c) evaluated the impact of releases on code smell survival. However, this study did not assess the impact of releases on the introduction and removal of code smells. Releases are usually considered as a factor that favours code smells and technical debt in general, since they push developers to code rapidly and meet deadlines regardless of quality constraints (Tom et al., 2013). Besides, mobile apps are known for having more frequent releases and updates (McIlroy et al., 2016), which may contribute to the prevalence of mobile code smells. Considering these potential factors, it is important to analyse the impact of releases on the presence of mobile code smells.

Lack of qualitative analysis. In a previous work (Habchi et al., 2018), we interviewed developers to understand their usage of linters to anticipate performance bottlenecks in mobile apps. This study gave insights about the adequacy of static analysers as a solution for mobile code smells. Nonetheless, other facets of these code smells still require qualitative investigation. In particular, we lack knowledge about how do developers remove mobile code smells from the source code. This knowledge is important to:

- Assess developers' awareness of mobile code smells;
- Check whether developers refactor these code smells intentionally or not;
- Learn removal techniques from developers and aliment future studies about code smell refactoring.

In this article, we address these lacks by answering the following research questions:

- **RQ1:** How frequent and diffuse are mobile code smell introductions?
- **RQ2:** How do releases impact introductions and removals of mobile code smells?
- **RQ3:** How do developers remove mobile code smells?
- **RQ4:** Do developers refactor mobile code smells?

To answer these questions, we build on the artefacts of our previous works (Habchi et al., 2019a,c) to perform an empirical study where we leverage both quantitative and qualitative analyses to inspect introductions and removals of 8 types of Android code smells. Specifically, we analyse the evolution of 180*k* code smell instances to answer RQ1 and RQ2. Then, we manually explore 561 code smell removals to answer RQ3 and finally we interview 25 smell-removing developers to answer RQ4. The results of this study show that:

1. Regarding frequency and diffuseness, there is an important discrepancy between code smell types. *No Low Memory Resolver* and *Leaking Inner Class* are the most diffuse by affecting more than 80% of the activities and inner classes, respectively.
2. Releases do not have an impact on the introductions and removals of code smells in open-source Android apps.
3. 79% of code smell instances are removed through the change history. However, these removals are mostly caused by large source code removals that do not mention refactoring. Also, only 19% of developers who authored these removals confirmed that their actions were intentional refactoring.
4. Developers who are aware of Android code smells do not necessarily refactor them. The code smell *Init OnDraw* was recognized by 64% of the participants, but only 12% of them refactored it.
5. Developers who intentionally refactor code smells affirm that their actions were driven and assisted by built-in code analysis tools.
6. Developers who did not refactor Android code smells doubted their performance impact and the usefulness of their refactoring. Some developers also preferred to handle performance issues when they arise instead of anticipating them.

This study provides a comprehensible replication package (Habchi et al., 2019b), which includes the used tools and data analysis scripts, the extracted data, and the results of the qualitative analysis.

The remainder of this article is organized as follows. Section 2 explains the study design, while Section 3 reports on the results. Section 4 interprets and discusses these results, and Section 5 exposes the threats to validity. Finally, Section 6 analyses related works, and Section 7 concludes with our main findings.

2. Study design

To perform this study, we relied on the artefacts that we built in our previous works about mobile code smells. In particular, we leveraged the dataset of code smell history (Habchi et al., 2019a,c) to collect the necessary data for this study. Then, we followed different approaches to analyse this data and answer our research questions.

2.1. Dataset

In previous works, we created a dataset containing the history of mobile-specific code smells. This dataset was built by running Sniffer (Habchi and Veuiller, 2019) on a set of Android apps and tracking 8 mobile code smells. For self-containment purposes, we present in this section (i) Sniffer, (ii) the 8 code smells, and (iii) the contents of this dataset.

2.1.1. Sniffer

Sniffer is an open-source (Habchi and Veuiller, 2019) toolkit that tracks the full history of Android-specific code smells. It tackles many issues raised by the Git mining community by tracking branches and detecting renaming (Kovalenko et al., 2018). Sniffer builds the code smell history by following a three-step process. First, from the repository of the app understudy, it extracts the commits and other necessary metadata like branches, releases, and commit authors. In the second step, it analyses the source code of each commit separately to detect code smell instances. Finally, based on the code smell instances and the repository metadata, it tracks the history of each smell and records it in the output database.

The performance of Sniffer was manually validated using 384 commits randomly sampled from open-source Android apps. This validation showed that it can detect code smell introductions with F1-score of 0.97 and code smell removals with a score of 0.96.

2.1.2. Code smells

The dataset covers all the 8 types of Android-specific code smells that are detectable by Sniffer. These code smells are performance-oriented and they originate from the catalogues of Reimann et al. (2014) and Hecht (2017), Hecht et al. (2015a). Unlike other Android code smells, these 8 smells are objective—*i.e.*, they either exist in the code or not, and cannot be introduced or removed gradually. Hence, their introduction and removal can be attributed to specific commits without confusion. Table 1 presents these code smells with a highlight on source code entities in which they can appear. We also mention the performance resource impacted by each code smell.

Table 1
Studied code smells.

Leaking Inner Class (*LIC*): in Android, anonymous and non-static inner classes hold a reference of the containing class. This can prevent the garbage collector from freeing the memory space of the outer class even when it is not used anymore, and thus causing memory leaks (Android, 2017; Reimann et al., 2014). **Entity**: Inner class. **Impact**: Memory.
Member Ignoring Method (*MIM*): this smell occurs when a method that is not a constructor and does not access non-static attributes is not static. As the invocation of static methods is 15%–20% faster than dynamic invocations, the framework recommends making these methods static (Hecht et al., 2015a). **Entity**: Method. **Impact**: CPU.
No Low Memory Resolver (*NLMR*): this code smell occurs when an `Activity` does not implement the method `onLowMemory()`. This method is called by the operating system when running low on memory in order to free allocated and unused memory spaces. If it is not implemented, the operating system may kill the process (Reimann et al., 2014). **Entity**: Activity. **Impact**: Memory.
Hashmap Usage (*HMU*): the usage of `HashMap` is inadvisable when managing small sets in Android. Using HashMaps entails the auto-boxing process where primitive types are converted into generic objects. The issue is that generic objects are much larger than primitive types, 16 and 4 bytes, respectively. Therefore, the framework recommends using the `SparseArray` data structure that is more memory-efficient (Android, 2017; Reimann et al., 2014). **Entity**: Method. **Impact**: Memory.
UI Overdraw (*UIO*): a UI Overdraw is a situation where a pixel of the screen is drawn many times in the same frame. This happens when the UI design consists of unneeded overlapping layers, *e.g.*, hidden backgrounds. To avoid such situations, the `canvas.quickreject()` API should be used to define the view boundaries that are drawable (Android, 2017; Reimann et al., 2014). **Entity**: View. **Impact**: GPU.
Unsupported Hardware Acceleration (*UHA*): in Android, most of the drawing operations are executed in the GPU. Rare drawing operations that are executed in the CPU, *e.g.*, `drawPath` method in `android.graphics.Canvas`, should be avoided to reduce CPU load (Hecht, 2017; Ni-Lewis, 2015). **Entity**: Method. **Impact**: CPU.
Init OnDraw (*IOD*): *a.k.a.* DrawAllocation, this occurs when allocations are made inside `onDraw()` routines. The `onDraw()` methods are responsible for drawing `Views` and they are invoked 60 times per second. Therefore, allocations (*init*) should be avoided inside them in order to avoid memory churn (Android, 2017). **Entity**: View. **Impact**: Memory.
Unsuited LRU Cache Size (*UCS*): in Android, a cache can be used to store frequently used objects with the *Least Recently Used* (LRU) API. The code smell occurs when the LRU is initialized without checking the available memory via the `getMemoryClass()` method. The available memory may vary considerably according to the device so it is necessary to adapt the cache size to the available memory (Hecht, 2017; McAnlis, 2015). **Entity**: Method. **Impact**: Memory.

2.1.3. Content

Running Sniffer on a set of 324 open-source Android apps resulted in a dataset with the history of all code smell instances that appeared in these apps. Table 2 summarizes the contents of this dataset. It is worth noting that the number of files presents all the files (classes, XML, configuration, etc.) that were analysed through the change history and not only the last versions of the apps. The first commit in this dataset is from November 2007 and the last one is from October 2018.

Table 2
Content of the dataset.

Apps	Commits	Files	Smell Instances	Developers	Releases	Branches
324	255,798	190,745	180,013	5,104	11,118	21,210

2.2. Data analysis

In this subsection, we describe our approach for analysing the collected data to answer our research questions. Table 3 reports on the list of metrics that we defined for this purpose.

As shown in Table 1, every code smell type affects a specific entity of the source code. Therefore, to compute the metric %diffuseness, we only focused on these entities. For instance, the code smell *Init OnDraw* affects only the entity `View`, thus we compute the percentage of views affected. This allows us to focus on the relevant parts of the source code and have a precise vision about the code smell diffuseness. For each app a, the diffuseness of a type of code smells t that affects an entity e is defined by:

$$\%\text{diffuseness}(a, t) = \frac{\#\text{affected-entities}(a, t)}{\#\text{available-entities}(a, e)}$$

For instance, the diffuseness of the code smell *No Low Memory Resolver (NLMR)* in an app a is:

$$\%\text{diffuseness}(a, NLMR) = \frac{\#\text{NLMR-instances}(a)}{\#\text{activities}(a)}$$

Where $\#\text{NLMR-instances}(a)$ is the number of *No Low Memory Resolver* instances in the app a and $\#\text{activities}(a)$ is the number of activities in a.

For the metrics #code-removed and %code-removed, we tracked the source code modifications that led to code smell removals. In particular, we counted all code smell removals where the host entity was also removed. For example, when an instance of the code smell *No Low Memory Resolver* is removed, the removal can be counted as #code-removed only if the host `Activity` has also been removed in the same commit.

2.2.1. RQ 1: How frequent and diffuse are mobile code smell introductions?

To inspect the prevalence of code smells, we computed—for each code smell type—the metrics: #introductions and %affected-apps. These metrics allow us to compare the prevalence of different code smell types. Then, to obtain a precise assessment of this prevalence, we also used the metric: %diffuseness. We computed the diffuseness of each code smell type in every app of our dataset. Finally, we plotted the distribution to show how diffuse are code smells compared to their host entities.

2.2.2. Rq 2: How do releases impact introductions and removals of mobile code smells?

This research question focuses on the impact of releases on code smell evolution. To ensure the relevance of this investigation, we paid careful attention to the suitability of the studied apps for a release inspection. In particular, we manually checked the timeline of each app to verify that it publishes releases through all the change history. We excluded apps that did not use releases at all, and apps that used them only at some stage. For instance, the Chanu app (Nittner, 2016) only started using

Table 3
Study metrics.

	Metric	Description
Code smell type	#introductions	The number of instances introduced in the dataset.
	%affected-apps	The percentage of apps affected by the code smell.
	%diffuseness	The diffuseness of the code smell instances in the source code of an app.
	#removals	The number of instances removed in the dataset.
	%removals	The percentage of instances removed—*i.e.*, $\frac{\#removals}{\#introductions}$.
	#code-removed	The number of instances removed with source code removal.
	%code-removed	The percentage of instances removed with source code removal—*i.e.*, $\frac{\#code-removed}{\#removals}$.
Commit	#commit-introductions	The number of code smell instances introduced by the commit.
	#commit-removals	The number of code smell instances removed by the commit.
	distance-to-release	The distance between the commit and the next release in terms of number of commits.
	time-to-release	The distance between the commit and the next release in terms of number of days.

releases in the last 100 commits, while the first 1337 commits do not have any releases. Hence, this app is, to a large extent, release-free and thus irrelevant for this research question. Out of the 324 studied apps, we found 156 that used releases during all the change history. The list of these apps can be found in our study artefacts (Habchi et al., 2019b). It is also worth noting that as Android apps are known for continuous delivery and releasing (Android, 2019; McIlroy et al., 2016), we considered in this analysis both minor and major releases. This allows us to perform a fine-grained study with more releases to analyse.

We used this set of 156 apps to evaluate the impact of releases on code smell introductions and removals. First, we visualized for each project the evolution of source code and code smells along with releases. We also plotted the evolution of code smell diffuseness for all studied apps. This visualization provides insights into the impact of releases and the evolution patterns of code smells.

To accurately measure the impact of releases, we analysed the effect of approaching releases on the numbers of introductions and removals performed in commits. Therefore, we used the metrics distance-to-release and time-to-release.

Distance to release. We aimed to evaluate the relationship between the distance to release and the numbers of code smells introduced and removed per commit. For this purpose, we assessed the correlation between the distance-to-release and both #commit-introductions and #commit-removals using Spearman's rank coefficient. Spearman is a non-parametric measure that assesses how well the relationship between two variables can be described using a monotonic function. This measure is adequate for our analysis as it does not require the normality of the variables and does not assess the linearity.

Time to release. Using the metric time-to-release, we extracted three commit sets:

- Commits authored 1 day before a release,
- Commits authored 1 week before a release,
- Commits authored 1 month before a release.

Then, we compared the #commit-introductions and #commit-removals in the three sets using Mann–Whitney U and Cliff's δ. We used the two-tailed Mann–Whitney U test (Sheskin, 2003) with a 99% confidence level, to check if the distributions of introductions and removals are identical in the three sets. To quantify the effect size of the presumed difference between the sets, we used Cliff's δ (Romano et al., 2006). Cliff is a non-parametric effect size measure, which is reported to be more robust and reliable than Cohen's d (Cohen, 1992). Moreover, it is suitable for ordinal data and it makes no assumptions of a particular distribution (Romano et al., 2006). For interpretation, we followed the common guidelines: negligible (N) for $|d| < 0.10$, small (S) for $0.10 \leq |d| < 0.33$, medium (M) for $0.33 \leq |d| < 0.474$, and large (L) for $|d| \geq 0.474$ (Grissom and Kim, 2005).

2.2.3. RQ 3: How do developers remove mobile code smells?

Quantitative analysis. First, we computed for each code smell type the metrics: #removals and %removals. Then, to gain insights about the actions that lead to code smell removals, we computed: #code-removed and %code-removed. The metric %code-removed reports the percentage of code smell instances that were removed with source code removal. This metric provides us a first idea about code smell removal techniques. To push further and identify the fine-grained actions that removed code smells, we opted for qualitative analysis.

Qualitative analysis. The objective of our analysis is to understand how code smells are removed. To achieve this, we manually analysed a sample of code smell removals. We used a stratified sample to make sure to consider a statistically significant sample for each code smell. In particular, we randomly selected a set of 561 code smell removals from our dataset. This represents a 95% statistically significant stratified sample with a 10% confidence interval of the 143,995 removals detected in our dataset. The stratum of the sample is represented by the 8 studied code smells. This sample includes commits from After sampling, we analysed every smell-removing commit to inspect two aspects:

- **Commit action:** The source code modification that led to the removal of the code smell instance. In this aspect, every code smell type has different theoretical ways to remove it. We inspect the commits to identify the actions used in practice for concretely removing code smells from the codebase;
- **Commit message:** We checked the messages looking for any mention of code smell removal. In this regard, we were aware that developers could refer to the smell without explicitly mentioning its name. Therefore, we thoroughly read the commit messages to look for implicit mentions of the code smell removal.

2.2.4. RQ 4 : Do developers refactor mobile code smells?

The objective of this question is to verify if the code smell removals detected in the change history are actual refactoring operations. For this purpose, we randomly selected 424 smell-removing developers, —*i.e.*, developers who performed code smell removals. This represents 30% of the 1414 smell-removing developers identified in our dataset. Afterwards, we filtered out developers that did not set proper emails in their git commits and we ended up with 340 developers that we can contact. We sent

emails to these developers to ask about the removed code smells. In particular, we presented the concerned code smell with the definition and code snippet that illustrates it. Then, we asked them the following questions:

1. Were you aware of this code smell?
2. Did you refactor this code smell intentionally?

The objective of the first question is to capture the developer's knowledge and awareness of the code smell. The second question allows us to check if the code smell removals authored by the developer are intended refactorings. Depending on the outcome of the second question, we asked one of the following questions:

3. Why did you refactor this code smell?
4. Why did not you refactor this code smell?

These open-questions allow developers to express their thoughts about mobile code smells and explain their choices about refactoring.

We received answers from 25 developers, which represents a response rate of 7, 35%. This rate is expectedly low as we ask developers about multiple code smell instances that impose them some deeper investment to recall and understand. The participants answered about all studied code smells, except *Unsupported Hardware Acceleration* and *Unsuited LRU Cache Size*. None of the responding developers was involved with these two code smells, which were indeed rare in our dataset. While most of the respondents only answered by text, two developers showed an interest in the topic and we were able to perform online interviews with them. The interviews initially followed the same textual questions, but depending on developers' answers, we asked additional questions. Consequently, we were able to get more detailed answers, especially for the two open-questions.

We transcribed the interview recordings into text using a denaturalism approach, which allows us to focus on informational content while still keeping a "full and faithful transcription". Together, the interviews and the answers to our open questions formed material for qualitative inspection. To analyse this material, we followed the analytical strategy of Schmidt (2004), which is well adapted for open questions. In this analysis, we relied on the two semantic categories:

- The reasons why developers refactor code smells;
- The reasons why developers do not refactor code smells;

To encode our material, we read the developers' answers and we tried to identify passages that relate to these categories. Based on these passages, we formulated new sub-categories. In our case, a sub-category represents a new reason for refactoring or not the code smell. To avoid redundancy, these sub-categories will later be presented when we report the results of this research question.

3. Study results

This section reports on the results of our study. It is worth noting that, to facilitate the replication of this study, all the results presented here are included in our companion artefacts (Habchi et al., 2019b).

3.1. RQ 1 : How frequent are code smell introductions?

Table 4 reports on the number of code smells introduced and the percentage of apps affected.

The table shows that, in the 324 analysed apps, 180,013 code smell instances were introduced. This number reflects the widespread of code smells in Android apps. Nonetheless, not all code smells are frequently introduced. Indeed, the table shows a significant disparity between the different code smell types. The code smells *Leaking Inner Class* and *Member Ignoring Method* were introduced more than 70,000 times, while *Unsuited LRU Cache Size* and *Init OnDraw* were only introduced less than 100 times. These results highlight two interesting observations:

- The most frequently introduced code smells, *Leaking Inner Class* and *Member Ignoring Method*, are both about source code entities that should be static for performance optimization;
- The UI-related code smells (*UI Overdraw*, *Unsupported Hardware Acceleration*, and *Init OnDraw*) are among the least frequently introduced code smells.

Regarding affected apps, Table 4 shows that 99% of apps had at least one code smell introduction in their change history, which again highlights the widespread of the phenomenon. The table also shows that the disparity in introduction frequency is reflected in the percentage of affected apps as frequent code smells tend to affect more apps. However, we observe that having more instances does not always imply affecting more apps. In particular, *No Low Memory Resolver* is much less present than *Leaking Inner Class* and *Member Ignoring Method*, 4198 *vs.* 98,751 and 72,228, respectively. Yet, it affected more apps, 99% *vs.* 96% and 85%.

To obtain a clear vision about these disparities, we reported in Fig. 1 the diffuseness of code smells within their host entities in the studied apps. The figure shows that *No Low Memory Resolver* is the most diffuse code smell. At least 50% of the dataset apps had this code smell in all their activities, *median* = 100%. *Leaking Inner Class* is also very diffuse. In most of the apps, it affected more than 80% of inner classes. Code smells that are hosted by views are less diffuse. On average, 15% of the views are affected by *UI Overdraw*. As for *Init OnDraw*, generally, it only affected less than 10% of the views. Finally, code smells hosted by methods are the least diffuse. *Member Ignoring Method*, *HashMap Usage*, *Unsupported Hardware Acceleration*, and *Unsuited LRU Cache Size* are present in less than 3% of the methods. This low diffuseness is not surprising as the number of methods is very high.

These results show that some frequent code smells, like *Member Ignoring Method*, are not diffuse, they only impact a small proportion of their potential host entities. Yet, code smells that seem less frequent, like *UI Overdraw* and *Init OnDraw*, are more diffuse and affect a bigger proportion of entities.

Android code smells are not introduced and diffused equally. *No Low Memory Resolver* and *Leaking Inner Class* are the most diffuse, in average they impact more than 80% of the activities and inner classes, respectively.

3.2. RQ 2 : How do releases impact introductions and removals of mobile code smells?

In this section, we report on the results of our release analysis on the 156 apps that used releases regularly. For each app, we generated code smell evolution curves that can be found in our artefacts (Habchi et al., 2019b). Fig. 2a shows an example of these curves that depicts the evolution of the number of code smells and classes in the Seafile client app. The figure highlights the releases to show the changes in code smell numbers when approaching releases. From our manual examination of all the evolution curves, we did not observe any tendency of code smell increase or decline immediately before or after releases. Generally, the number of code smells evolves with an important growth at the first stages of feature development. Then, this growth stabilizes as the projects enter the maintenance phase. Naturally, this pattern is not followed by all the analysed

Table 4
Numbers of code smell introductions.

Code smell	*LIC*	*MIM*	*NLMR*	*HMU*	*UIO*	*UHA*	*IOD*	*UCS*	All
#introductions	98,751	72,228	4198	3944	514	267	93	18	180,013
%affected-apps	96	85	99	60	36	20	15	2	99

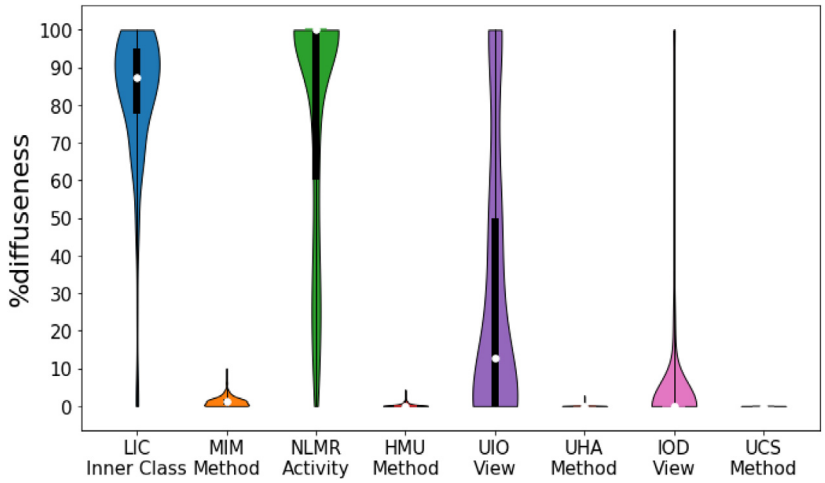

Fig. 1. Distribution of code smell %diffuseness in studied apps.

projects as in many cases some components or modules are removed, which results in a drop in the project size and the number of code smells. Fig. 2b presents an example of these cases that were observed in the Syncthing app. Regardless of the growth pattern, we observe that the number of code smells follows the project size in terms of number of classes. These observations align with Lehman's laws of continuing growth and declining quality where the increase in code smells is an indicator of declining quality.

To isolate the impact of project size, we also generated evolution curves for code smell diffuseness. Figs. 2c – 2f show examples of curves generated for four Android apps. From our inspection of these curves, we did not observe any impact of releases on code smell diffuseness. Sometimes, we notice abrupt drops or peaks in code smell diffuseness but these events are not explicitly related to releases. We also notice that the diffuseness evolution did not follow one simple pattern, like the raw number of code smells. However, based on general trends, we observed that three patterns were emerging frequently: consistent rise, consistent decline, and stability.

Fig. 2c shows an example of consistent rise in code smell diffuseness observed in the Subsonic project. We can see how the project started with 0.4 code smells per class and rose consistently to reach 0.8 code smells per class after 3600 commits. The opposite pattern is observed in Fig. 2d where code smell diffuseness declines over the lifetime of the AndStatus app. At the early stages of this project, the diffuseness was around 0.65 smells per class and it declined progressively and ended up around 0.4 smells per class. The KISS app, depicted in Fig. 2c, shows an example of stable code smell diffuseness. Despite some abrupt peaks and drops in the initial commits, code smell diffuseness always ranged between 0.35 and 0.45 all along 2800 commits. Some apps did not fall under any of these patterns and their code smell evolution had random changes along the project lifetime. For instance, the 4pdaClient app, had a hill-shaped evolution curve as shown in Fig. 2f. Indeed, the diffuseness evolved constantly in the first 200 commits, then started decreasing to go back to the same initial diffuseness.

Beyond this manual analysis, we assessed the impact of releases using the metrics distance-to-release and time-to-release.

3.2.1. Distance to release

Fig. 3 presents two scatter plots that show the relationship between the distance from releasing and the number of code smell introductions and removals per commit. The first thing that leaps to the eye is the similarity between the two plots. Code smell introductions and removals are similarly distributed regarding the distance from releasing. We do not notice any time window where the code smell introductions and removals are negatively correlated. We also do not visually observe any correlation between the distance from release and code smell introductions and removals. Indeed, the Spearman's rank correlation coefficients confirm the absence of such correlations.

$$Spearman(\text{distance-to-release}, \#\text{commit-introductions}) \times \begin{cases} \rho = 0.04 \\ p\text{-}value < 0.05 \end{cases}$$

$$Spearman(\text{distance-to-release}, \#\text{commit-removals}) \times \begin{cases} \rho = 0.01 \\ p\text{-}value < 0.05 \end{cases}$$

The results show that for both correlations, the *p-value* is below the threshold. Hence, we can consider the computed coefficients as statistically significant. As these correlation coefficients are negligible, we can conclude that there is no monotonic relationship between the distance from releasing and the numbers of introductions and removals per commit.

3.2.2. Time to release

After analysing the impact of the distance to release, we investigate the impact of the time to release on code smell introductions and removals.

Fig. 4 shows the density function of code smell introductions and removals in different timings. First, we observe that code smell introductions and removals are distributed similarly. For each timing, the density function of code smell introductions and removals are analogous.

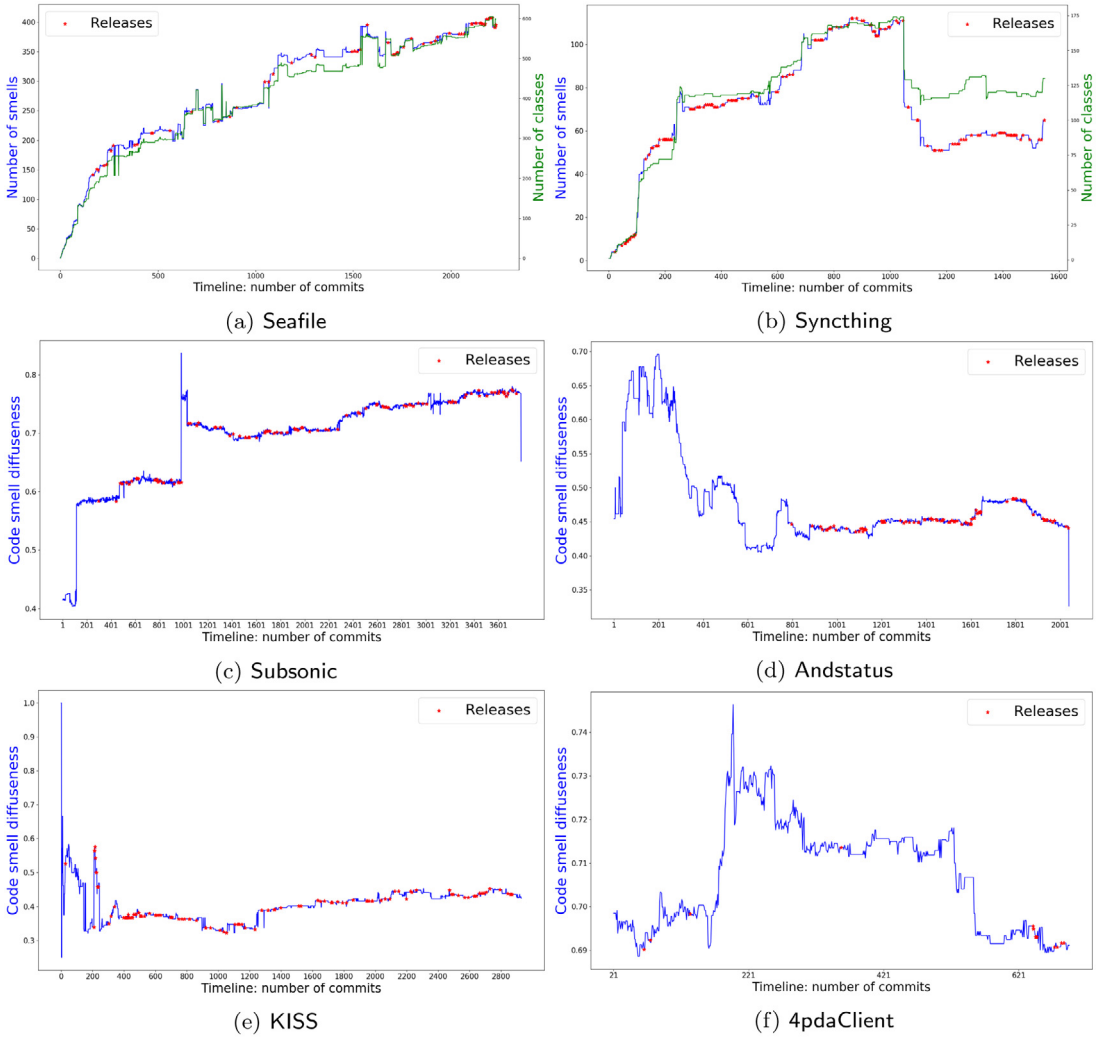

Fig. 2. The evolution of code smells in different Android apps.

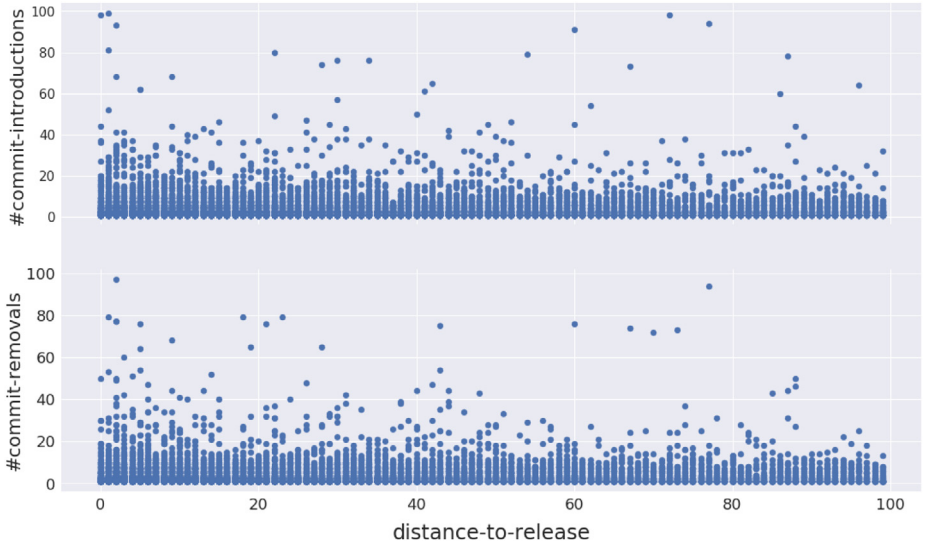

Fig. 3. The number of code smell introductions and removals per commit in the last 100 commits before release.

As for the comparison between code smell introductions performed at different times, we observe that commits performed one day before releasing have a higher probability to only have one code smell introduction. Commits performed one week or one month before release tend also to have around one code smell introduction, but they also have chances to introduce more code

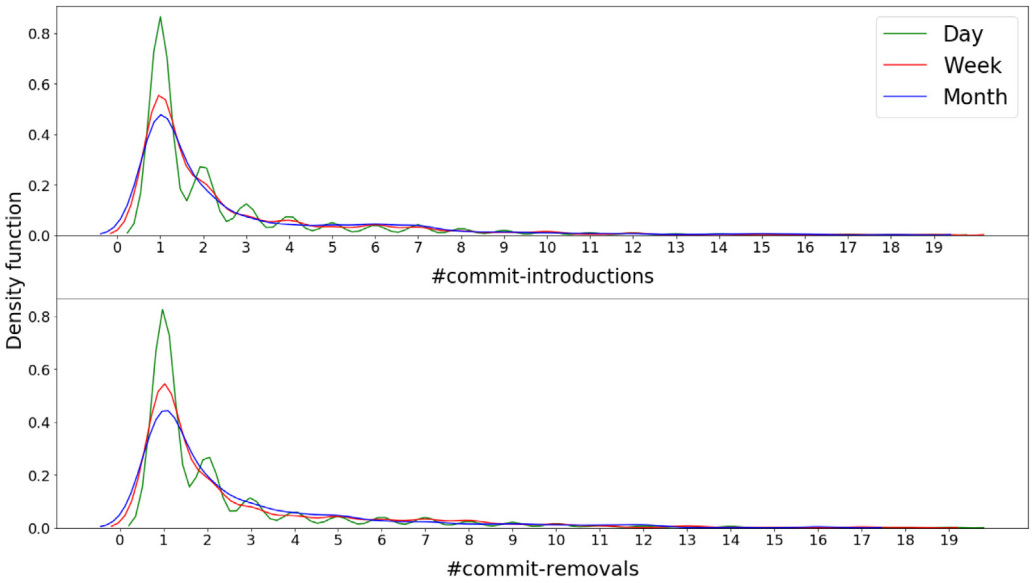

Fig. 4. The density function of code smell introductions and removals one day, one week, and one month before releasing.

smells. This means that commits authored one day before the release do not necessarily have more code smell introductions.

Code smell removals follow the same distribution for every timing. Thus, we can infer that time to release has no visible impact on code smell introductions and removals.

To confirm this observation, we compare in Table 5 the code smell introductions performed one day, one week, and one month before the release.

The table results show that for all code smells, there is no significant difference between code smell introductions occurring on different dates before the release ($p\text{-}value > 0.01$). The effect size values confirm the results, all the quantified differences are small or negligible.

Similarly, Table 6 compares code smell removals in commits authored one day, one week, and one month before the release. The results are similar to the ones observed for code smell introductions. The differences between different commits sets are insignificant ($p\text{-}value > 0.01$) and effect sizes are small or negligible regardless of the code smell type.

These observations suggest that there is no difference between the introduction and removal tendencies in commits authored just before release and those written days or weeks before. It is worth noting that for *UI Overdraw*, *Init OnDraw*, *Unsupported Hardware Acceleration*, and *Unsuited LRU Cache Size*, the number of instances was in some cases insufficient for performing the statistical tests. Hence, our results are not applicable to these code smells.

Releases do not have an impact on the introductions and removals of Android code smells.

3.3. RQ 3 : How do developers remove mobile code smells?

3.3.1. Quantitative analysis

Table 7 reports on the number and percentage of removals. The table shows that on average 79% of code smell instances are removed. Looking at every code smell type separately, the removal rate varies between 35% and 93%. *Member Ignoring Method* is the most removed code smell with 93% of its instances were removed along with the change history. On the other hand, *Unsupported Hardware Acceleration* is the least removed code smell with only 35% of its instances removed. The other code smells have a coherent removal percentage; they all had from 60% to 70% of their instances removed.

The table also reports the number and percentage of code smell instances removed within source code removal—*i.e.*, code-removed. The table shows that overall 59% of code smell removals are a result of removing source code. For all code smell types, except *Member Ignoring Method*, more than 50% of code smell removals are accompanied with the removal of their host entities. *Member Ignoring Method* is the only code smell that is rarely removed with source code removals— %code-removed = 20%.

Table 8 compares smell-removing commits with commits that did not remove code smell instances, in terms of number and percentage of code lines deleted. The table shows that 50% of smell-removing commits deleted at least 41 code lines and 25% of these commits deleted more than 1.39% of the codebase. This shows that code smell removals occur in commits that perform large code deletions. On average, smell-removing commits delete 10 times more lines than commits that do not remove code smells. A similar discrepancy is observed in the percentage of code deleted from the codebase. 50% of smell-removing commits deleted more than 0.24% of the codebase, whereas the same portion of non-removing commits deleted less than 0.029% of the codebase.

Overall, 79% of code smell instances are removed through the change history. Except for *Member Ignoring Method*, most of code smells are removed because of source code removal.

3.3.2. Qualitative analysis

Table 9 summarizes the results of our manual analysis of 561 smell-removing commits. For each code smell type, the table presents the number of analysed instances, a breakdown of the actions used to remove it, and the percentage of messages mentioning its removal. The following subsections report on these results in details. For some code smells, the removal actions were similar, thus we report them together. Also, for the sake of clarification, we remind for each code smell all actions that can be performed to remove it before reporting the actions found in the analysed sample.

Table 5
Compare #commit-introductions in commits authored one day, one week, and one month before releasing.

	LIC	*MIM*	*NLMR*	*HMU*	*UIO*	*IOD*	*UHA*	*UCS*	All
Day	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	–	$p > 0.01$
Week	0.01(N)	0.05(N)	0.08(N)	0.20(S)	0.18(S)	0.10(S)	–	–	0.00(N)
Day	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	–	–	–	–	$p > 0.01$
Month	0.02(N)	0.04(N)	0.08(N)	0.02(N)	–	–	–	–	0.01(N)
Week	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	–	–	–	–	$p > 0.01$
Month	0.04(N)	0.00(N)	0.00(N)	0.04(N)	–	–	–	–	0.01(N)

Table 6
Compare #commit-removals in commits authored one day, one week, and one month before releasing.

	LIC	*MIM*	*NLMR*	*HMU*	*UIO*	*IOD*	*UHA*	*UCS*	All
Day	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	–	$p > 0.01$
Week	0.05(N)	0.03(N)	0.02(N)	0.05(N)	0.29(S)	–	–	–	0.01(N)
Day	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	–	–	–	–	$p > 0.01$
Month	0.02(N)	0.07(N)	0.30(S)	0.01(N)	–	–	–	–	0.03(N)
Week	$p > 0.01$	$p > 0.01$	$p > 0.01$	$p > 0.01$	–	–	–	–	$p > 0.01$
Month	0.02(N)	0.04(N)	0.30(S)	0.04(N)	–	–	–	–	0.01(N)

Table 7
Number and percentage of code smell removals.

Code smell	*LIC*	*MIM*	*NLMR*	*HMU*	*UIO*	*IOD*	*UHA*	*UCS*	All
#removals	70,654	67,777	2526	2509	305	147	66	11	143,995
%removals	71	93	60	63	59	35	70	61	79
#code-removed	67,169	13,809	1625	1824	273	123	33	8	84,864
%code-removed	95	20	64	73	90	84	50	73	59

Table 8
Number and percentage of lines deleted in smell-removing commits.

Commit type	#Line deletions		%Line deletions	
	Smell-removing	Non-removing	Smell-removing	Non-removing
Q1	10	1	0,034	0.004
Median	41	3	0.246	0.029
Q3	133	13	1.394	0.152

Code smell: Leaking Inner Class (LIC).
Possible removals:
- Make the inner class static;
- Remove the inner class.

Commit actions: We found that in 98% of the cases, *LIC* instances are removed because inner classes are removed with other parts of the code. For instance, a commit from the Seadroid app that fixes bugs also removes unused code that contained a non-static inner class (GitHub, 2013). Hence, the commit has removed a *LIC* instance as a side effect of the bug fixing. This finding explains the high percentage of code-removed found for *LIC* in the quantitative analysis (95%).

We only found one case of *LIC* removal that was not caused by source code deletion. It was a commit that refactored a feature and made an inner class private and static, thus removing a code smell instance (GitHub, 2010). As this commit made diverse other modifications, we could not affirm that the action was an intended code smell refactoring.

Commit message: We did not find any explicit or implicit mention of *LIC* in the messages of smell-removing commits. Moreover, the messages did not refer specifically to the removed inner classes. Even the unique commit that removed a *LIC* instance with a modification did not mention anything about the matter in its message (GitHub, 2010).

Code smell: Member Ignoring Method (MIM).

Possible removals:
- Make the affected method static;
- Add method body, *i.e.*, introduce code that accesses non-static attributes to the affected method;
- Remove the affected method.

Commit actions: We found that only 15% of *MIM* removals are due to the deletion of the host methods. In most of cases, *MIM* was rather removed with the introduction of source code. Specifically, when empty methods are developed—with instructions added inside—they do not correspond to the *MIM* definition anymore, and thus code smell instances are removed. Also, other instances are removed from full methods with the introduction of new instructions that access non-static attributes and methods. Finally, we did not find any case of *MIM* removal that was performed by only making the method static.

Commit message: We did not find any commit message that referred to the removal of *MIM* instances.

Code smell: No Low Memory Resolver (NLMR).

Possible removals:
- Add the method `onLowMemory()` to the activity;
- Remove the affected activity.

Table 9
Results of manual analysis.

Code smell	#Instances	Commit actions	Message
LIC	96	Remove inner-class (98%) Make inner-class static (2%)	0%
MIM	96	Add method body (85%) Remove Method (15%)	0%
NLMR	93	Remove Activity (65%) Transform Activity (35%)	0%
HMU	74	Remove Method (70%) Remove statements (30%)	0%
UIO	74	Remove method (88%) Remove statements (11%) Add `clipRect()` (1%)	0%
IOD	40	Remove method (55%) Remove statements (45%)	15%
UHA	59	Remove method (85%) Remove statements (15%)	0%
UCS	11	Remove method (91%) Remove statements (9%)	0%

Commit actions: We found that 65% of *NLMR* instances are removed with source code deletion. This deletion is caused by large modifications in the code base, like major migrations and the addition of new features. For instance, a commit from the Silence app refactored the whole app to start using `Fragment` components (GitHub, 2012a). One consequence of these modifications is the deletion of the `SecureSMS` activity, which used to be an instance of *NLMR*.

As for the remaining 35% instances, the removal was due to the conversion of host activities into other components. For example, a commit from the K-9 app converts an activity that was an instance of *NLMR* into a fragment (GitHub, 2012b). As the code smell *NLMR* is about activities, the class as fragment does not correspond to the definition anymore and thus the code smell instance is removed. Other than these two actions, we did not find any other ways of removing *NLMR* instances. In particular, we did not find any case where the method `onLowMemory()` is added to refactor the code smell.

Commit message: We did not find any commit message that mentioned the removal of *NLMR* instances. We found one message that mentions that the commit performs memory improvement in two classes (GitHub, 2009). Nonetheless, these improvements were not related to the *NLMR* code smell.

Code smells: HashMap Usage (HMU), Unsupported Hardware Acceleration (UHA) & Init OnDraw (IOD).

Possible removals:
- Remove the statements that introduced them;
- Remove their host entities.

Commit actions: In the manual analysis of these code smells, we inspected whether the removal was due to the deletion of large code chunks or only the removal of the specific statements that caused the instance. We found that the instances of *HMU* and *UHA* are usually removed with the deletion of their host methods, 70% and 85% respectively. As for *IOD*, there are equally instances removed with big code deletions as instances removed with only statement removals. Looking for potential intended refactorings, we carefully examined the cases where instances are removed at a low granularity level—*i.e.*, statements. In all *HMU* and *UHA* instances, we did not find a code smell removal that could represent an intended refactoring. All the instances are removed as a side effect of modifications inside methods that do not specifically target the code smell statement. However, we found that 22% of *IOD* instances are removed with precise modifications that sound like proper refactoring. Indeed, there are 9 *IOD* instances that specifically removed the `init` statement or extracted it out of the `onDraw()` method. Another element that incite us to describe these modifications as intended is that they removed the linter warning suppression of *DrawAllocation*. This shows that the developers were aware of removing a code smell instance.

Commit message: Out of the 9 potential proper refactorings of *IOD*, we found 6 commit messages that mentioned the code smell removal. This confirms that the operations are intended refactorings. As for *HMU* and *UHA*, none of the analysed messages mentioned their removal.

Code smells: UI Overdraw (UIO) & Unsuited LRU Cache Size (UCS).

Possible removals:
- Add method calls:
  - `clipRect()` or `quickReject()` for *UIO*;
  - `getMemoryClass()` for *UCS*;
- Remove the code smell statements;
- Remove the host methods.

Commit actions: We found that most of *UIO* and *UCS* instances are removed with other parts of the source code, 88% and 91% respectively. The remainder instances were removed with modifications inside methods that implied the deletion of code smell statements. The only exception was one *UIO* instance, which was removed with the introduction of a call to the method `clipRect()`. This modification was the only case that sounded like a proper refactoring.

Commit message: None of the analysed messages mentioned the code smells *UIO* and *UCS*.

Android code smells are usually removed with large source code removing commits that do not mention refactoring. In our dataset, *Init OnDraw* and *UI Overdraw* are the only code smells that were subject to apparent refactoring and only *Init OnDraw* was mentioned in commit messages.

3.4. RQ 4 : Do developers refactor mobile-specific code smells?

Fig. 5 shows the answers collected for our two first questions. The blue sub-bars present the percentage of developers that affirmed their awareness or refactoring of the code smell. The orange sub-bars present the percentage of developers answering with maybe, while the grey ones present the percentage of developers answering with No. We observe that on average mobile code smells were recognized by 43% of the participants. *Leaking Inner Class*, *Init OnDraw*, and *HashMap Usage* are the most acknowledged code smells. More than 56% of the participants were aware of them. *UI Overdraw* was acknowledged by 48% of participants, whereas *Member Ignoring Method* and *No Low memory Resolver* were recognized by only 36% of the participants.

When it comes to refactoring, we notice a drop by approximately 50% from developers that already recognized code smells. Indeed, on average, only 19% of the participants affirmed that they refactored mobile code smells. Also, *Leaking Inner Class* and *HashMap Usage*, which were recognized by more than 50% of the participants, were only refactored by 32% and 28% of them. The same drop is observed for *Member Ignoring Method* and *UI*

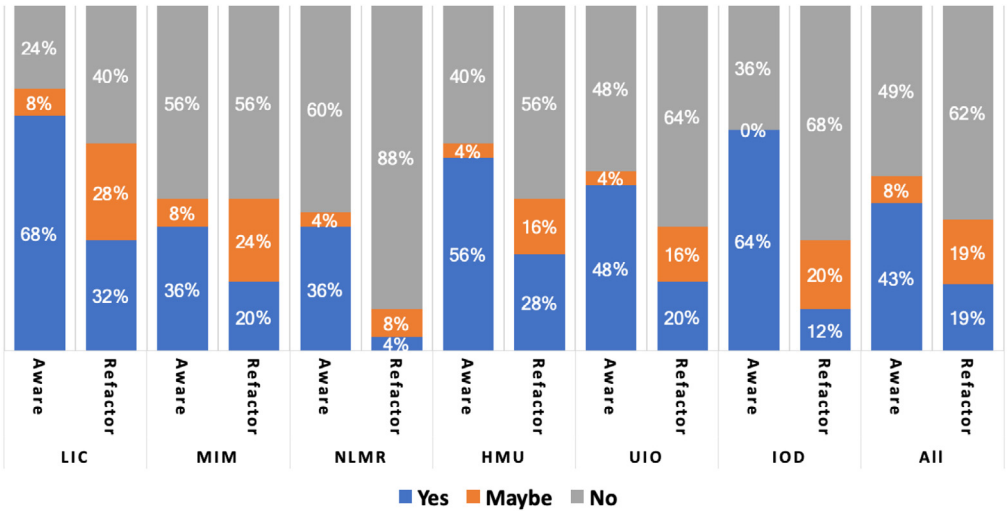

Fig. 5. Answers about code smell awareness and refactoring. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Overdraw, which were refactored by only 20% of the participants. Interestingly, *No Low Memory Resolver* and *Init OnDraw* were rarely refactored, 4% and 12% respectively. This proportion is particularly low considering that *Init OnDraw* was acknowledged by 64% of the participants.

Another observation is the large proportion of uncertain respondents for the refactoring questions—*i.e.*, developers responding with maybe. Indeed, this proportion ranged between 4% and 8% for the awareness question, whereas it ranged between 8% and 28% for the refactoring question. This is reasonable as the participants may not remember for sure if they have already refactored this code smell or not, thus the uncertain answer.

On average, 43% of the participants recognized mobile code smells but only 19% of them confirmed their refactoring. Developers who are aware of Android code smells do not necessarily refactor them. *Init OnDraw* was recognized by 64% of the participants and only refactored by 12% of them.

For the open-questions, we present the answers following the semantic sub-categories that we identified.

3.4.1. Reasons why developers refactor mobile code smells

Code analysis tools. Three participants claimed that they refactor code smells because they are reported by code analysis tools. One participant affirmed refactoring all critical code smells that are detected by Android Lint. *"I trust the default configuration of the linter, so if something is flagged as critical, I will stop the build to fix it"*. Another participant emphasized the impact of using such tools, *"IDEs and their built-in code analysis tools provide good warnings on these problems, which encourage people to fix them, even if they are unaware of them"*. According to another participant, the help given by these tools goes beyond refactoring code smells, *"the tools provide context and explanation that sharpen the programmers' perception in the future to avoid such problems beforehand. This kind of nudging should not be underestimated"*.

Personal practices. Two developers considered code smell refactoring as a good development practice that they adopt. The first participant said that she refactors code smells regularly, *"I always try to improve the code source of projects that I work on. If I notice an issue, I fix it"*. The second participant described refactoring code smells as a part of the routine that she follows while getting into an existing code base, *"for me it is a practice to dig into foreign code and while doing so, fixing smells along the way, as I gain understanding of the code base"*. This developer explained that these smells are easy to refactor as they do not change from an app to another, *"they do not relate to the app specifically, they are common knowledge"*.

Freedom. One participant affirmed that she was able to perform refactoring operations only because she was the main maintainer of an open-source project with no external stakeholders. *"I have full control over releases. I do not have external pressure, so I can invest time in keeping the source code at the quality level that satisfies me"*. Interestingly, this developer explicitly claims that this same freedom was not available in industrial projects that she contributed to as part of her job.

Developers who refactored Android code smells are motivated and assisted by built-in code analysis tools and their personal commitment to code quality.

3.4.2. Reasons why developers do not refactor mobile code smells

The impact is not significant. Five participants judged that the impact of mobile code smells is not big enough to care about them. In particular, one developer claimed that *"the performance difference is really tiny or non-existent in the end"*. The same developer also claimed that most of these code smells are automatically mitigated by the runtime, *"ART can perform this kind of optimizations"*. Another developer explained how the architecture of mobile apps makes these code smells less concerning, *"well designed apps have most of the logic implemented in the backend and unless it is a game, the UI is not updated very often. Therefore, most of these performance issues are usually no issue at all"*. A similar argument was constructed by another developer who believed that performance issues arise from the connection to the backend instead of the frontend code smells, *"network latency typically dominates in mobile app responsiveness"*. Other participants gave specific examples about code smells that seemed impotent for them. One developer downplayed the impact of *HashMap Usage*, claiming that *"using SparseArray collections is better for memory usage but it is less important on modern devices than it was ten years ago"*. *No Low Memory Resolver* was also considered irrelevant by one participant who described different approaches to free memory using the activity life-cycle callbacks, *e.g.*, `onPause()` and `onResume()`. This participant also added that this code smell

is not commonly acknowledged by developers, *"I have been part of the Android community since the very beginning, 2010, and I do not recall hearing about this code smell at all"*.

Not a performance problem. Three participants expressed their doubts about the relationship between mobile code smells and performance. For instance, one developer considered that *Member Ignoring Method* is a "code usability" issue but not a performance one. Another developer estimated that these code smells are *"not directly related to performance"* giving as example *No Low Memory Resolver* and *Leaking Inner Class*. Another developer believed that these issues should be qualified as *"code quality, completeness, correctness, resource usage issues"* instead of performance.

Refactoring would not help. Two developers judged that the refactoring of code smells is useless by giving as example *No Low Memory Resolver*. Specifically, one developer considered that receiving `No Low Memory` warnings is a sign of bad memory management by the app, and responding to the system warning would not fix the issue. She explained: *"if that point is reached it probably means you have a memory leak or another problem with the way your app manages memory and clearing some cache to prevent an out of memory crash is just a temporary band-aid before the app eventually crashes anyway"*. Another participant went further by considering that refactoring *No Low Memory Resolver* could even lead to the introduction of new bugs. She believes that *"handling low memory seem more likely to do harm than good as implementations are likely to be buggy and of little benefit compared to letting the app be killed"*.

Performance issues are better handled reactively. Two participants considered that developers should not worry about these code smells because performance issues should not be handled proactively. The first participant stated: *"gut feelings about performance issues are usually wrong"*. Thus, she advises against trusting these feelings or instincts, *"never try to optimize before a profiling has shown where exactly the problem in your application is"*. The other participant gave a similar advice and recommended relying on reactive tools like profilers to deal with performance issues when they arise, *"instead of worrying about details I advise Android developers to worry more about UX and if the app performance slows just use the profiler to locate the bottlenecks and fix that"*.

Prioritization. Two participants mentioned that refactoring mobile code smells is not a high priority. One developer referenced the perpetual trade-off between quality improvement and new features, *"I always have a panoply of ideas for improving my codebase but I learned to prioritize features that directly benefit the client"*. The developer also described different criteria that she considers while initiating a refactoring, *"I evaluate considering the source code quality in the long term. If the refactoring does not help in terms of maintenance and performance, I will not perform it"*.

The practice is justifiable. One participant judged that some practices that were labelled as code smells were justifiable. She claimed that the use of `HashMaps` is a good and necessary practice because the alternative data structure makes code maintenance worse. She explained this by stating that *"using Android framework `SparseArray` classes in a component prevents it from being tested in JVM unit tests"*.

Developers that did not refactor Android code smells doubt their performance impact and the usefulness of their refactoring. Some developers also prefer to handle performance issues when they arise instead of anticipating them.

4. Discussion and implications

Releases. The results of RQ2 show that the pressure of releases do not have an impact on code smell introductions and removals. This suggests that code smells are not introduced as a result of releasing pressure. Moreover, when asked about the reasons for not refactoring code smells, developers did not blame releases. One developer mentioned prioritization, but this did not include releases and rather explained how different quality aspects and outcomes are prioritized. These results may challenge the common beliefs about releases and their relationship with technical debt in general (Tom et al., 2013). However, it is noteworthy that our results are based on open-source projects, which can be different from their industrial counterparts. This point was raised by the participant who praised the freedom and control that she had in her open-source project and who was aware that such circumstances are rare in industrial projects. Hence, we encourage future studies to:

- Evaluate the impact of releases on mobile code smells in industrial projects and ecosystems.

Awareness of code smells. Previous studies suggested that the accrual of mobile code smells and the indifference of developers toward it are signs of unawareness (Habchi et al., 2019a,c). However, the inputs collected from developers in RQ4 challenge this hypothesis. Our participants claimed to recognize many Android code smells, but they had other reasons to neglect them. In particular, some developers were reluctant to code smell refactoring because they assumed that it would lead to further issues. To remove this obstacle, we encourage researchers and toolmakers to:

- Build tools that propose automated refactoring of mobile code smells.

Static analysis tools. We observed that code smells that are detected by Android Lint—*i.e.*, *Leaking Inner Class*, *Init OnDraw*, *HashMap Usage*, and *UI Overdraw*—are the most recognized by developers (Android, 2017). Developers who performed refactoring also explained that their actions were motivated and assisted by built-in code analysis tools. Also, the only apparent refactorings identified in RQ3 was for *Init OnDraw* and *UI Overdraw*, which are detected by Android Lint. Some of these refactorings explicitly deleted Android Lint suppressions, which shows that developers considered the linter warnings and responded with an intended refactoring. These findings confirm that static analysers can help in raising awareness about code smells and refactoring them. Hence, we encourage researchers and toolmakers to:

- Build and integrate static analysers with more code smell coverage.

Removal and refactoring. The quantitative and qualitative findings of RQ3 show that code smells are mainly removed with source code deletion. Even for *Member Ignoring Method* instances, which are removed with source code introduction, we found that they are removed because the empty and primitive methods are developed with new statements. While we cannot judge the intentions of a source code modification, most of the analysed commits did not reveal signs of intended refactoring and did not mention the code smell. On top of that, the answers collected in RQ4 indicate that only a minority of smell-removing developers did perform a refactoring. Hence, we can suggest that most of code smell removals are a side effect of other maintenance activities and are not intentional refactorings. This implies that:

- We cannot rely on code smell removals to learn refactoring techniques. Future studies that intend to learn from the

change history to build automated refactoring tools cannot rely on the removals of these code smells as learning examples.

Controversial code smells. According to RQ4, *No Low Memory Resolver* is the least acknowledged and refactored code smell, 36% and 4% respectively. This code smell was disapproved by many developers who explained that the absence of a resolver does not systematically result in memory issues and its presence is not always useful. This disapproval can explain why this code smell affected 99% of the studied apps and was the most diffuse of our 8 code smells. Questions were also raised about other code smells, like UI code smells, which were downplayed, and *HashMap Usage*, which was described as justifiable and irrelevant in modern devices. Following these questions, we invite future research works to:

- Reassess the relevance of these code smells and check the accuracy of their definitions.

Manage performance reactively. Developers explained that instead of worrying about code smells, they prefer handling performance bottlenecks when they arise. This reactive approach was already observed and discussed in previous studies about mobile apps (Habchi et al., 2018; Linares-Vasquez et al., 2015), yet the research contributions in this area remain rare. Specifically, many static analysers were provided to detect performance issues in mobile apps (Habchi et al., 2017; Hecht et al., 2015b; Palomba et al., 2017) and, to the best of our knowledge, no profiler was provided to help in managing bottlenecks when they appear. For this reason, we encourage future works to:

- Build profilers that can help developers in spotting performance bottlenecks and identifying their root causes.

Relationship between code smells and performance bottlenecks. Previous studies showed that mobile developers look after performance and take bottlenecks seriously (Linares-Vasquez et al., 2015), yet when asked about code smells developers seem less preoccupied. In particular, our participants doubted the impact of code smells and questioned their association with performance. This shows that some developers do not perceive a causal relationship between code smells and performance bottlenecks. Indeed, the existence of such a relationship remains theoretical. Previous studies relied on repeated execution scenarios to demonstrate the impact of Android code smells on performance (Carette et al., 2017; Hecht et al., 2016; Palomba et al., 2019), but they did not associate them with bottlenecks. Therefore, we encourage future studies to:

- Study the relationship between mobile code smells and performance bottlenecks.

5. Threats to validity

General threats. The main threat to our internal validity could be an imprecise detection of code smell introductions and removals. This imprecision is relevant in situations where code smells are introduced and removed gradually, or when the change history is not accurately tracked. However, this study only considered objective code smells that can be introduced or removed in a single commit. As for history tracking, we relied on Sniffer, which tracks branches and renamings and accurately detects code smell introductions and removals (F1-score $= \{0.97, 0.96\}$). As for external validity, the main threat is the representativeness of our results. We used a dataset of 324 open-source Android apps with 255*k* commits and 180*k* code smell instances. It would have been preferable to consider also closed-source apps to build a more diverse dataset. However, we did not have access to any proprietary software that can serve this study. We encourage future studies to consider other datasets of open-source apps to extend this study (Geiger et al., 2018; Krutz et al., 2015). We also encourage the inclusion of apps of different sizes as the frequency of mobile code smells follows the codebase size. Another possible threat to external validity is that our study only concerns 8 Android-specific code smells. Without further investigation, these results should not be generalized to other code smells or development frameworks. We, therefore, encourage future studies to replicate our work on other datasets and with different code smells and mobile platforms.

RQ 2. A possible threat to internal validity is the selection of releases and projects. We avoided this threat by selecting apps that had releases all along with their change history. This measure ensures the accuracy of the metrics distance-to-release and time-to-release. Another potential threat for this question could be the presence of app releases that are not marked on GitHub. Our release detection relies on GitHub tags, which are designed for this purpose, but developers can always release their projects without using such tags. Hence, it is possible for our approach to miss some app releases. Furthermore, our results about the impact of releases on code smells are limited to open-source apps. Apps developed as part of industrial projects can be subject to more external requirements and releasing pressure. We encourage future works to extend our work by inspecting the impact of releases in different settings.

RQ 3. One possible threat to the internal validity of our results could be the accuracy of our manual analysis. We tried to alleviate this threat by relying on objective criteria like the actions performed by the commit and the content of its message. We also did not judge the intentions of developers and counted on their answers in RQ4 to assess the proportion of real refactoring. Another threat could be the generalizability of the results of our qualitative analysis. We used a randomly selected set of 561 smell-removing commits. This represents a 95% statistically significant stratified sample with a 10% confidence interval of the 143,995 removals detected in our dataset. To support the credibility of our study, we also provide this set with our study artefacts.

RQ 4. The results of our user study can be threatened by the sampling bias. We sent our questions to a set of 340 smell-removing developers because our objective was to check if their removals were actual refactoring operations. Without further investigation, the observed proportions of awareness and refactoring cannot be generalized to all mobile developers. Furthermore, the answers collected in this study may be subject to acquiescence and desirability biases. Participants may be inclined to answer with "yes" to agree with us or seem more aware of software quality issues. We minimized these biases by keeping the answers anonymous and avoiding the implication that some answers are "right" or "wrong". We also allowed participants to express their points of view through the open-questions.

6. Related works

In this section, we report on the literature related to code smells in mobile apps and their analysis in the change history.

6.1. Mobile code smells

The first reference to mobile-specific code smells was when Reimann et al. (2014) proposed a catalogue of 30 quality smells dedicated to Android. These code smells originate from the good and bad practices presented online in Android documentation.

They cover various aspects like implementations, user interfaces, or database usages and they are reported to harm properties, such as efficiency, user experience, or security. Many research works built on this catalogue and proposed approaches and tools for detecting code smells in mobile apps (Habchi et al., 2017; Hecht et al., 2015b; Kessentini and Ouni, 2017; Palomba et al., 2017). In particular, Hecht et al. (2015a) proposed Paprika, a tooled approach that detects OO and Android smells in Android apps. Paprika models Android apps as a large architectural graph and queries it to detect code smells. Palomba et al. (2017) proposed another tool, called aDoctor, able to identify 15 Android-specific code smells from the catalogue of Reimann et al. Habchi et al. (2017) proposed an extension of Paprika that detects iOS-specific code smells. Lately, Gupta et al. (2019) used 3 machine learning algorithms to generate rules that detect four Android code smells. In their experiments, the JRip algorithm achieved the best results by generating rules capable of detecting smells with a 90% overall precision.

To cope with mobile code smells, researchers also proposed refactoring solutions. Lin and Dig (2015) proposed Asynchronizer, a tool that extracts long-running operations into AsyncTasks, and AsyncDroid, a tool that transforms improperly-used `AsyncTasks` into Android `IntentService`. Morales et al. (2017) proposed EARMO, an energy-aware refactoring approach for mobile apps. They identified the energy cost of 8 OO and mobile antipatterns. Based on the cost, EARMO generates refactoring sequences automatically.

6.2. Empirical studies on mobile code smells

Most empirical studies focused on assessing the performance impact of mobile code smells on app performance (Carette et al., 2017; Hecht et al., 2016; Morales et al., 2016; Palomba et al., 2019). Hecht et al. (2016) conducted an empirical study about the individual and combined impact of 3 Android smells. They measured the performance of 2 apps with and without smells using the following metrics: frame time, number of delayed frames, memory usage, and number of garbage collection calls. The measurements showed that refactoring the *Member Ignoring Method* smell improves the frames metrics by 12.4%. Carette et al. (2017) studied the same code smells, but focused on the energy impact. They analysed 5 open-source Android apps and observed that in one of them the refactoring of the 3 code smells reduced the global energy consumption by 4, 83%. The study of Morales et al. (2017) also showed by analysing 20 open-source apps that refactoring antipatterns can decrease significantly energy consumption of mobile apps. Notably, Palomba et al. (2019) showed that methods that represent a co-occurrence of *Internal Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow Loop*, consume 87 times more energy than other smelly methods.

Beyond the performance impact, empirical studies compared the distribution of code smells in mobile apps and desktop systems. Specifically, Mannan et al. (2016) compared the presence of OO code smells in Android apps and desktop applications. They did not observe major differences between these two types of applications in terms of density of code smells. However, they found that the distribution of OO code smells in Android is more diversified than for desktop applications. Further, Habchi et al. (2017) analysed 279 iOS apps and 1500 Android apps to compare the presence of OO and mobile-specific smells in the two platforms. They observed semantic similarities between the code smells exhibited by the two platforms. On top of that, they found that Android apps tend to have more OO and mobile-specific code smells.

While these studies helped in understanding the distribution and impact of mobile-specific code smells, they did not provide any qualitative insights about the topic. Indeed, the only study that leveraged qualitative analysis is the one from Habchi et al. (2018), which investigated the perception of performance bad practices by Android developers. This study reported that developers may lack interest and awareness about Android code smells. Moreover, the study showed that some developers challenge the relevance and impact of code smells in practice. Our work complements this study as it relies on another information source—removal instances—to understand the phenomenon of code smells in practice. Besides, our work also provides quantitative insights into how these code smells are introduced and removed in practice.

6.3. Code smells in the change history

The evolution of code smells through the change history has been addressed by various studies in the OO context. Tufano et al. (2017) addressed questions similar to our study. They, analysed the change history of 200 open-source projects to understand when and why code smells are introduced and for how long they survive. They observed that most of code smells instances are introduced when files are created and not due to evolution process. They also found that new features and enhancement activities are responsible for most smell introductions, and newcomers are not necessarily more prone to introducing new smells. Interestingly, this study also investigated the rationales of code smell removal, showing that only 9% of code smells are removed with specific refactoring operations. Our study yields similar results as it shows that even though 79% of code smell instances are removed through the change history, only 19% of code smell removers described their actions as intentional refactoring.

Peters and Zaidman (2012) conducted a case study on 7 open-source systems to investigate the lifespan of code smells and the refactoring behaviour of developers. They found that, on average, code smell instances have a lifespan of approximately 50% of the examined revisions. Moreover, they noticed that, usually, one or two developers refactor more than the others, however, the difference is not large. Finally, they observed that the main refactoring rationales are cleaning up dead or obsolete code, dedicated refactoring, and maintenance activities.

Tufano et al. (2016) analysed the change history of 152 open source projects to inspect the evolution of test smells and their relationship with code smells. Their results showed that, similarly to OO code smells, test smells are introduced when tests are created and they have a high survivability. Their results also suggest the existence of relationship between test smells and code smells of the code under test.

In the context of mobile apps, we have already leveraged the change history to study code smells in previous works (Habchi et al., 2019a,c). The first work studied developer contributions showing that the ownership of code smells is spread across developers regardless of their seniority and experience. As for the second one, it studied code smell survival and showed that while in terms of time Android code smells can remain in the codebase for years before being removed, it only takes 34 effective commits to remove 75% of them. These results suggested that developers lack interest in code smells and most of their actions toward them are accidental. Result-wise, our study complements these findings as it shows the reasons behind developers' inaction toward code smells. Novelty-wise, our study relies on the artefacts of these works to address new topics:

- Removal fashions: This study inspects the actions that lead to code smell removals;
- Refactoring: In this study, we discuss with developers to (i) check if they intentionally refactor code smells and to (ii) identify the motivations behind their actions;

- Releases and diffuseness: Our previous work evaluated the impact of releases on code smell survival (Habchi et al., 2019c). In this work, we go further and assess the impact of releases on code smell introductions and removals. On top of that, we provide insights about the evolution of code smells and their diffuseness.

Another relevant study for our work was conducted by Mazuera-Rozo et al. (2020) who manually analysed 500 commits that fixed performance bugs in Android and iOS apps. This analysis allowed them to build a taxonomy of performance bugs and confirm that GUI lagging, energy leak, and memory bloat are the most common performance bugs in mobile apps. The study also analysed the survival of performance bugs showing that on average they remain for at least 90 days, which surpasses the average lifetime of other bug types.

7. Conclusion

We presented in this article a large-scale empirical study that leverages quantitative and qualitative analyses to improve our understanding of mobile code smells. The main findings of this study are:

- **Diffuseness**: Android code smells are not introduced and diffused equally. *No Low Memory Resolver* and *Leaking Inner Class* are the most diffuse by affecting 90% of activities and inner classes.
- **Releasing pressure**: Releases do not have an impact on the frequency of code smell introductions and removals in open-source Android apps;
- **Removal**: 79% of code smell instances are removed through the change history. However, these removals are mostly caused by large source code removals that do not mention refactoring. Also, only 19% of developers who authored these removals confirmed that their actions were intentional refactorings;
- **Awareness**: Developers who are aware of Android code smells do not necessarily refactor them. The code smell *Init OnDraw* was recognized by 64% of the participants, but only 12% of them refactored it;
- **Refactoring:** Developers who refactored Android code smells were motivated and assisted by built-in code analysis tools and their commitment to code quality. On the other hand, developers that did not refactor Android code smells doubted their performance impact and the usefulness of their refactoring. Some developers also preferred to handle performance issues when they arise instead of anticipating them.

These findings have notable implications on future research agenda:

- We encourage future works to evaluate the impact of releases on mobile code smells in industrial projects and ecosystems. This need arises from the remarks of developers about the contrast between the freedom that they have while developing open-source apps and the pressure that they undergo in industrial projects;
- Future studies that intend to learn from the change history to build automated refactoring tools cannot rely on code smell removals as learning examples;
- To address the questions and doubts raised by developers, we need to reassess the relevance of Android code smells and check the accuracy of their definitions;
- We intend to study the relationship between mobile code smells and performance bottlenecks to understand and assess their impact on performance.

Besides, based on our findings, we encourage tool makers to:

- Build profilers that can help developers in spotting performance bottlenecks and identifying their root causes rapidly;
- Build and integrate static analysers with more code smell coverage. This is beneficial as we observed the impact of such tools on developer awareness and actions;
- Build tools that propose automated refactoring of mobile code smells. Such tools are crucial for developers who are reluctant toward refactoring by fear of introducing further issues.

This study also provides a comprehensive replication package (Habchi et al., 2019b), which includes the tools, datasets, and results.

CRediT authorship contribution statement

Sarra Habchi: Conceptualization, Methodology, Investigation, Writing - original draft. **Naouel Moha:** Conceptualization, Writing - review & editing, Funding acquisition. **Romain Rouvoy:** Conceptualization, Writing - review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Android, 2017. Android lint checks. [Online; accessed April-2021], https://sites.google.com/a/android.com/tools/tips/lint-checks.

Android, 2019. Android versioning. [Online; accessed January-2019], https://developer.android.com/studio/publish/versioning.

Carette, A., Younes, M.A.A., Hecht, G., Moha, N., Rouvoy, R., 2017. Investigating the energy impact of android smells. In: Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on. IEEE, pp. 115–126.

Cohen, J., 1992. A power primer.. Psychol. Bull. 112 (1), 155.

Geiger, F.-X., Malavolta, I., Pascarella, L., Palomba, F., Di Nucci, D., Bacchelli, A., 2018. A graph-based dataset of commit history of real-world android apps. In: Proceedings of the 15th International Conference on Mining Software Repositories. ACM, pp. 30–33.

GitHub, 2009. Mention memory improvement. [Online; accessed January-2019], https://github.com/k9mail/k-9/commit/909f677f912ed1a01b4ef39f2bd7e6b068d1f19e.

GitHub, 2010. Fix LIC. [Online; accessed January-2019], https://github.com/connectbot/connectbot/commit/32bc0edb89e708b873533de94d3e58d5099cc3ba.

GitHub, 2012a. Remove NLMR. [Online; accessed January-2019], https://github.com/SilenceIM/Silence/commit/3d9475676f80a3dbd1b29f83c59e2c132fb135b5.

GitHub, 2012b. Remove NLMR with modifications. [Online; accessed January-2019], https://github.com/k9mail/k-9/commit/bbcc4988ba52ca5e8212a73444913d35c23cebc4.

GitHub, 2013. Remove LIC. [Online; accessed January-2019], https://github.com/haiwen/seadroid/commit/74112f7acba3511a650e113aa3483dcd215af88f.

Grissom, R.J., Kim, J.J., 2005. Effect sizes for research: A broad practical approach.. Lawrence Erlbaum Associates Publishers.

Gupta, A., Suri, B., Bhat, V., 2019. Android smells detection using ML algorithms with static code metrics. In: International Conference on Recent Developments in Science, Engineering and Technology. Springer, pp. 64–79.

Habchi, S., Blanc, X., Rouvoy, R., 2018. On adopting linters to deal with performance concerns in android apps. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. In: ASE 2018, ACM, New York, NY, USA, pp. 6–16. http://dx.doi.org/10.1145/3238147.3238197.

Habchi, S., Hecht, G., Rouvoy, R., Moha, N., 2017. Code smells in iOS apps: How do they compare to android? In: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems. IEEE Press, pp. 110–121.

Habchi, S., Moha, N., Rouvoy, R., 2019a. The rise of android code smells: Who is to blame? In: Proceedings of the 16th International Conference on Mining Software Repositories. In: MSR '19, IEEE Press, Piscataway, NJ, USA, pp. 445–456. http://dx.doi.org/10.1109/MSR.2019.00071.

Habchi, S., Moha, N., Rouvoy, R., 2019b. Study artifacts. [Online; accessed June-2019], https://figshare.com/s/790170a87dd81b184b0a.

Habchi, S., Rouvoy, R., Moha, N., 2019c. On the survival of android code smells in the wild. In: Proceedings of the 6th International Conference on Mobile Software Engineering and Systems. In: MOBILESoft '19, IEEE Press, Piscataway, NJ, USA, pp. 87–98, URL: http://dl.acm.org/citation.cfm?id=3340730.3340749.

Habchi, S., Veuiller, A., 2019. Sniffer source code. [Online; accessed March-2019], https://github.com/HabchiSarra/Sniffer/.

Hecht, G., 2017. Détection et analyse de l'impact des défauts de code dans les applications mobiles (Ph.D. thesis). Université du Québec à Montréal, Université de Lille, INRIA.

Hecht, G., Moha, N., Rouvoy, R., 2016. An empirical study of the performance impacts of android code smells. In: Proceedings of the International Workshop on Mobile Software Engineering and Systems. ACM, pp. 59–69.

Hecht, G., Omar, B., Rouvoy, R., Moha, N., Duchien, L., 2015a. Tracking the software quality of android applications along their evolution. In: 30th IEEE/ACM International Conference on Automated Software Engineering. IEEE, p. 12.

Hecht, G., Rouvoy, R., Moha, N., Duchien, L., 2015b. Detecting Antipatterns in Android Apps. Research Report RR-8693, INRIA Lille ; INRIA, URL: https://hal.inria.fr/hal-01122754.

Kessentini, M., Ouni, A., 2017. Detecting android smells using multi-objective genetic programming. In: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems. IEEE Press, pp. 122–132.

Kovalenko, V., Palomba, F., Bacchelli, A., 2018. Mining file histories: should we consider branches? In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, pp. 202–213.

Krutz, D.E., Mirakhorli, M., Malachowsky, S.A., Ruiz, A., Peterson, J., Filipski, A., Smith, J., 2015. A dataset of open-source android applications. In: Proceedings of the 12th Working Conference on Mining Software Repositories. IEEE Press, pp. 522–525.

Lin, Y., Dig, D., 2015. Refactorings for android asynchronous programming. In: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on. IEEE, pp. 836–841.

Linares-Vasquez, M., Vendome, C., Luo, Q., Poshyvanyk, D., 2015. How developers detect and fix performance bottlenecks in android apps. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 352–361.

Mannan, U.A., Ahmed, I., Almurshed, R.A.M., Dig, D., Jensen, C., 2016. Understanding code smells in android applications. In: Proceedings of the International Workshop on Mobile Software Engineering and Systems. ACM, pp. 225–234.

Mazuera-Rozo, A., Trubiani, C., Linares-Vásquez, M., Bavota, G., 2020. Investigating types and survivability of performance bugs in mobile apps. Empir. Softw. Eng. 1–43.

McAnlis, C., 2015. The magic of LRU cache (100 days of google dev). [Online; accessed January-2019], https://youtu.be/R5ON3iwx78M.

McIlroy, S., Ali, N., Hassan, A.E., 2016. Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. Empir. Softw. Eng. 21 (3), 1346–1370.

Morales, R., Saborido, R., Khomh, F., Chicano, F., Antoniol, G., 2016. Anti-patterns and the energy efficiency of android applications. arXiv preprint arXiv:1610.05711.

Morales, R., Saborido, R., Khomh, F., Chicano, F., Antoniol, G., 2017. EARMO: An energy-aware refactoring approach for mobile apps. IEEE Trans. Softw. Eng..

Ni-Lewis, I., 2015. Custom views and performance (100 days of google dev). [Online; accessed January-2019], https://youtu.be/zK2i7ivzK7M.

Nittner, G., 2016. Chanu - 4chan android app. [Online; accessed January-2019], https://github.com/grzegorznittner/chanu.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., De Lucia, A., 2017. Lightweight detection of android-specific code smells: The adoctor project. In: Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on. IEEE, pp. 487–491.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., De Lucia, A., 2019. On the impact of code smells on the energy consumption of mobile applications. Inf. Softw. Technol. 105, 43–55.

Peters, R., Zaidman, A., 2012. Evaluating the lifespan of code smells using software repository mining. In: Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on. IEEE, pp. 411–416.

Reimann, J., Brylski, M., Aßmann, U., 2014. A tool-supported quality smell catalogue for android developers.. Softw. Trends 34 (2), URL: http://dblp.uni-trier.de/db/journals/stt/stt34.html#ReimannBA14.

Romano, J., Kromrey, J.D., Coraggio, J., Skowronek, J., 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the NSSE and other surveys. In: Annual Meeting of the Florida Association of Institutional Research. pp. 1–33.

Schmidt, C., 2004. The analysis of semi-structured interviews. A Compan. Qual. Res. 253–258.

Sheskin, D.J., 2003. Handbook of parametric and nonparametric statistical procedures. crc Press.

Tom, E., Aurum, A., Vidgen, R., 2013. An exploration of technical debt. J. Syst. Softw. 86 (6), 1498–1516.

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

Tufano, M., Palomba, F., Oliveto, R., Penta, M.D., Lucia, A.D., Poshyvanyk, D., 2017. When and why your code starts to smell bad (and whether the smells go away). IEEE Trans. Softw. Eng. PP, http://dx.doi.org/10.1109/TSE.2017.2653105.