# The Rise of Android Code Smells: Who Is to Blame?

Sarra Habchi, Naouel Moha, Romain Rouvoy

# The Rise of Android Code Smells: Who Is to Blame?

Sarra Habchi
*Inria / University of Lille*
Lille, France
sarra.habchi@inria.fr

Naouel Moha
*Université du Québec à Montréal*
Montréal, Canada
moha.naouel@uqam.ca

Romain Rouvoy
*University of Lille / Inria / IUF*
Lille, France
romain.rouvoy@inria.fr

***Abstract*—The rise of mobile apps as new software systems led to the emergence of new development requirements regarding performance. Development practices that do not respect these requirements can seriously hinder app performances and impair user experience, they qualify as code smells. Mobile code smells are generally associated with inexperienced developers who lack knowledge about the framework guidelines. However, this assumption remains unverified and there is no evidence about the role played by developers in the accrual of mobile code smells. In this paper, we therefore study the contributions of developers related to Android code smells. To support this study, we propose SNIFFER, an open-source toolkit that mines Git repositories to extract developers' contributions as code smell histories. Using SNIFFER, we analysed 255k commits from the change history of 324 Android apps. We found that the ownership of code smells is spread across developers regardless of their seniority. There are no distinct groups of code smell *introducers* and *removers*. Developers who introduce and remove code smells are mostly the same.**

***Index Terms*—Android, mobile apps, code smells, history mining**

## I. INTRODUCTION

Since the emergence of mobile applications (apps) as new mainstream software systems, researchers were interested in bad development practices in these frameworks, *a.k.a.* code smells. Mobile-specific code smells are different from the well known *object-oriented* (OO) code smells as they often refer to a misuse of the mobile platform SDK and they are more performance-oriented [44]. According to the guidelines of mobile frameworks, they hinder memory, CPU, and energy performances [3], [4], [36], [37]. This theoretical assumption has been confirmed by empirical studies that assessed the impact of mobile-specific code smells on app performances [9], [27], [39]. Hence, mobile-specific code smells represent *"a gap between the current state of a software system and some hypothesized ideal state in which the system is optimally successful in a particular environment"*, which is a definition of technical debt [8].

Technical debt is a well founded concept in software engineering and, since its definition by Cunningham *et al.* [12], researchers and practitioners have been investigating its rationales. Today, it is well established that developers attitude, ignorance, and oversight are some of its major drivers [10], [15], [48]. These three drivers are particularly relevant in the context of Android code smells. As these code smells tend to be oriented towards performance issues, inexperienced developers who do not have an extensive understanding of the framework may not be able to guess them intuitively. For instance, the code smell *UI Overdraw* manifests when a `Canvas` is used without defining the boundaries with `clipRect()` or `quickReject()` [4], [43]. A developer who does not master the drawing process of Android cannot guess intuitively that these particular methods should be called while drawing. Furthermore, as many mobile developers have been working on desktop software beforehand, they may still adopt desktop practices without realising their bad impact on apps performance. A concrete example of these practices are *HashMap Usage* and *Member Ignoring Method* which are accepted practices in Java, but considered as code smells in Android.

Interestingly, none of the previous studies on mobile code smells has analysed the developers angle. They only approached this topic in their explanatory hypotheses when discussing the possible factors of mobile code smells accumulation. In particular, they hypothesised that some mobile developers who lack experience and knowledge about the framework, are responsible for the accrual of mobile code smells [18], [19], [26]. Our study builds on these hypotheses and addresses the developers role with a large scale analysis. Indeed, understanding this role is crucial for explaining the phenomenon of mobile code smells and proposing efficient solutions to tackle them.

Specifically, we aim at answering the following research questions.

**RQ1:** How do developers contribute in code smell introductions and removals?

This question is important for quantifying developers interest and awareness about Android code smells. If developers who introduce and remove code smells are isolated and distinct, this would show that the unawareness about code smells is not prevalent and only restrained to some developers and *vice versa*. In this topic, the literature and the common understanding of code smells provide three main hypotheses.

**Hypothesis 1 (H1):** *Code smell introductions and removals are isolated practices.*

This hypothesis is supported by studies that suggested that the presence of code smells is the result of the actions of only some developers [18], [19].

**Hypothesis 2 (H2):** *There is an inverse correlation between code smell introductions and removals among developers.*

This hypothesis is motivated by the reasoning that some *bad developers* who are unaware about code smells will tend to introduce and never remove them and *vice versa* [18], [19], [26], [29].

**Hypothesis 3 (H3):** *For each code smell type, there is an inverse correlation between code smell introductions and removals among developers.*

This hypothesis aligns with the previous one but goes deeper by considering the knowledge about every code smell separately.

**RQ2:** How does seniority impact developers contribution?

This question allows us to investigate whether code smell unawareness is caused by inexperience. This information is necessary to define the developers who should be targeted firstly while designing tools and documentations about code smells. The suggested hypotheses for this question are the following.

**Hypothesis 4 (H4):** *Newcomers tend to introduce more and remove less code smells.*

This hypothesis relies on the conventional wisdom that inexperienced developers who are new to the project are bad contributors.

**Hypothesis 5 (H5):** *Reputable developers tend to introduce less and remove more code smells.*

Reputable developers have a status in the community and they are considered as good contributors [14]. This hypothesis suggests that, as good developers, they should introduce less and remove more code smells.

Our paper provides three notable contributions. It presents the first large-scale empirical study that investigates the role of developers in the accrual of mobile-specific code smells. Our study analyses 8 Android code smells, 324 Android apps, $255k$ commits, and contributions from $4,525$ developers. Our results show that the accrual of mobile-specific code smells is not due to the actions of isolated groups of developers. This paper also proposes SNIFFER [22], a novel open-source toolkit that accurately mines the change history of Android apps to track developers' contributions. SNIFFER tackles many issues raised by the Git mining community like branch and renaming tracking. Finally, we provide all the collected data for this paper in a database that contains the history of $180.013$ Android code smells [21].

The remainder of this paper is organised as follows. Section II explains the study design, while Section III reports on the results. Section IV interprets and discusses these results, and Section V presents related works. Section VI concludes with our main learned lessons.

## II. Study Design

We start this section with a presentation of our study context. Afterwards, we introduce our novel toolkit SNIFFER. Then, we conclude with a description of our approach for analysing the extracted data to answer our research questions.

### A. Context Selection

The core of our study is analysing the change history of mobile apps to inspect developers behaviour regarding code smells. In that respect, the context selection entails the choice of (1) the mobile platform to study, and (2) the mobile-specific code smells with their detection tool.

*1) The Mobile Platform:* We decided to focus our study on the Android platform. With $85.9\,\%$ of the market share, Android is the most popular mobile operating system as of 2018.[1] Moreover, more Android apps are available in open-source repositories compared to other mobile platforms [19]. On top of that, both development and research communities proposed tools to analyse the source code of Android apps and detect their code smells [4], [29], [38].

*2) Code Smells & Detection Tool:* In the academic literature, the main reference to Android code smells is the catalog of Reimann *et al.* [42]. It includes 30 code smells, which are mainly performance-oriented, and covers various aspects, like user interface and data usage. Ideally, we would consider all the 30 code smells in our study. However, for feasibility, we could only consider code smells that are already detectable by state-of-the-art tools. Hence, the choice of studied code smells will be determined by the adopted detection tool. In this regard, our detection relied on PAPRIKA, an open-source tooled approach that detects Android-specific code smells from Android packages (APK). PAPRIKA is able to detect 13 Android-specific code smells. However, after examination we found that two of these code smells, namely *Invalidate Without Rect* and *Internal Getter Setter*, are now deprecated [3], [5]. Thus, we excluded them from our analysis. Moreover, we wanted to focus our study on objective code smells—*i.e.*, smells that either exist in the code or not, they cannot be introduced or removed gradually. Hence, we excluded *Heavy AsyncTask*, *Heavy Service Start* and *Heavy BroadcastReceiver*, which are subjective code smells [29]. We present in Table I brief descriptions of the eight code smells that we kept for our study.

### B. Dataset & Selection Criteria

To select the apps eligible for our study, we relied on the famous FDROID online repository[2]. This choice allowed us to include published Android apps and exclude dummy apps, templates, and libraries that could be available on GitHub. We automatically crawled the apps available on FDROID and retrieved their GitHub links when available. Then using these links, we reached the repositories on GitHub. For computational constraints and as we focus on developers, we only kept repositories which had at least two developers. This filter resulted in $324$ projects with $255,798$ commits and $4,525$ developers. The full list of projects can be found in our artifacts [20].

[1]https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems

[2]https://f-droid.org

TABLE I
STUDIED CODE SMELLS.

| |
|---|
| **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 [28]. |
| **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 [3]. |
| **No Low Memory Resolver (NLMR)**: this code smell occurs when an `Activity` does not implement the `onLowMemory()` method. This method is called by the system when running low on memory in order to free allocated and unused memory spaces. If it is not implemented, the system may kill the process [43]. |
| **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 [3], [43]. |
| **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 which is more memory-efficient [3], [43]. |
| **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.*, hiding backgrounds. To avoid such situations the method `clipRect()` or `quickReject()` should be called to define the view boundaries that are drawable [4], [43]. |
| **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 [26], [37]. |
| **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 initialised without checking the available memory via the method `getMemoryClass()`. The available memory may vary considerably according to the device so it is necessary to adapt the cache size to the available memory [26], [36]. |

## *C.* SNIFFER

We designed and developed SNIFFER [22], an open-source, heavily tested, and documented toolkit that tracks the full history of Android-specific code smells. Figure 1 shows an overview of the SNIFFER process. We detail in the following subsections each step of this process.

*1) Step 1: Extract App Data:*
**Input:** Git link to an Android app.
**Output:** Commits and repository model.
This step is performed by the sub-module `CommitLooper` [22]. First, SNIFFER clones the repository from Git and parses its log to obtain the list of commits. Afterwards, it analyses the repository to extract its model. This model consists of properties of different repository elements like commits, developers, branches, etc. While most of the properties are directly extracted from Git, some others need more processing to be retrieved. In particular, the SDK version is retrieved from the `Manifest.xml` file, which describes the configuration of the Android app. Also, properties related to branches are not directly available. Branches are a local concept in Git, thus information about the original branch of a commit is not recorded to be easily retrieved. For this, SNIFFER makes an additional analysis to attribute each commit to its original branch. In particular, it navigates the commit tree by crossing all commit parents and extracts the repository branches. This extraction allows us to track the smell history accurately in step 3.

*2) Step 2: Detect Code Smells:*
**Input:** Commits and repository model.
**Output:** Code smell instances per commit.
This step is performed by the sub-module `SmellDetector` [22]. As explained beforehand, SNIFFER relies on PAPRIKA to detect Android code smells. Originally, PAPRIKA detects code smells from the APK, it does not analyse the source code. However, we wanted to detect code smells directly from the source code of commits. Therefore, we needed to integrate a static analyser into SNIFFER that feeds PAPRIKA with the necessary data. In such way, before going through smell detection with PAPRIKA, each commit goes through the static analysis.

*Static analysis :* SNIFFER performs static analysis using SPOON [40], a framework for Java-based programs analysis and transformation. SNIFFER launches SPOON on the commit source code to build an abstract syntax tree. Afterwards, it explores this tree to extract code entities (*e.g.*, classes and methods), properties (*e.g.*, names, types), and metrics (*e.g.*, number of lines, complexity). Together, these elements constitute a usable model by PAPRIKA.

*Detection of code smell instances :* Fed with the model built by the static analyser, PAPRIKA detects the code smell instances. To this point, commits are still processed separately. Thus, this step produces a separate list of code smell instances for each commit.

*3) Step 3: Analyse Change History:*
**Input:** Code smell instances per commit and the repository model.
**Output:** Full code smell history.
This step is performed by the sub-module `SmellTracker` [22]. In this step, SNIFFER tracks the full history of every code smell. As our study focuses on objective Android code smells, we only have to look at the current and previous commits to detect smell introductions and removals. If a code smell instance appears in a commit while absent from the previous, a smell introduction is detected. In the same way, if a commit does not exhibit an instance of a smell that appeared previously, a smell removal is detected. In order for this process to work, SNIFFER needs to (a) retrieve the previous commits accurately and (b) track renamings.

*Retrieve the previous commit accurately:* Retrieving the previous commits requires an order in the commit history. In this regard, Git log provides two main commit orders:

- Chronological order by date or authoring date;
- Topological order.

These two orders flatten the commit tree into one branch and this implies a loss of accuracy. To illustrate this effect,

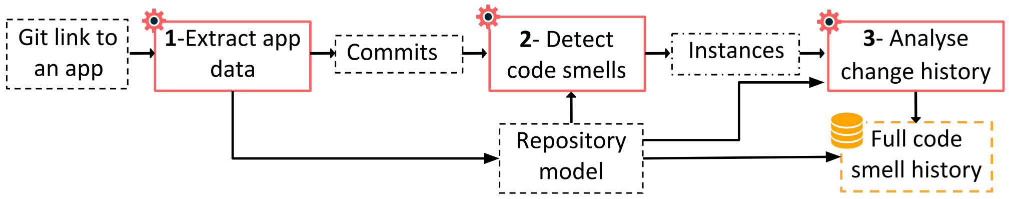

Fig. 1. Overview of the SNIFFER toolkit.

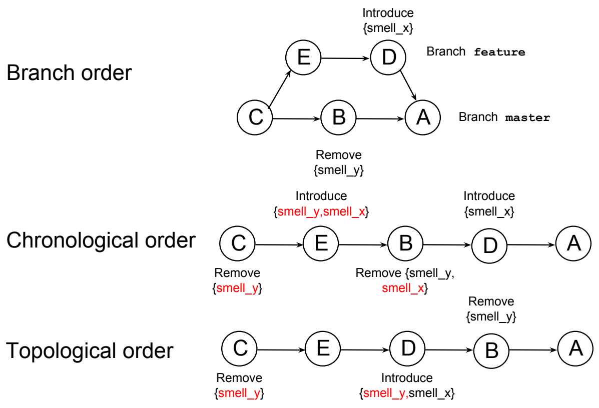

Fig. 2. Commits ordered in different settings.

we depict in Figure 2 an example of a commit tree with two branches. The branch order presents the real commit history where the commit D introduced smell_x and commit B removed smell_y. With the chronological order, commit B is placed between commits D and E. With the topological order, commits D and E are placed between *B* and *C*. Moreover, in both orders the commit C has only one previous commit instead of two. As shown in Figure 2, these placements caused many false code smell introductions and removals. This shows that these orders are suitable for repositories with a single branch. However, for multiple branches, we need to follow a branch-aware order. For this purpose, SNIFFER considers the branches extracted in step 1 to retrieve previous commits. This allows to stay faithful to the real order of commits and accurately detect code smell introductions and removals.

*Track renamings :* In the process of development, files and folders can be renamed. These renamings hinder the tracking of code smells. To prevent these mistakes, SNIFFER relies on Git to detect all the renamings happening in the repository. Git tracks files with their contents instead of their names or paths. Hence, if a file or a folder is renamed and the content is not drastically changed, Git is able to keep track of the file. Git uses a similarity algorithm [2] to compute a similarity index between two files. By default, if the similarity index is above 50 %, the two files are considered the same. SNIFFER uses this feature via the command `git log --find-renames --summary`, which shows all the renamings that happened in the repository. SNIFFER uses these detected renamings to link the files and code smells in the output database. Consequently, when a renaming occurs, no new code smell introductions or removals are detected, and the history of code smells is accurately tracked.

The generated full code smell history is saved in a PostgreSQL database that can be queried to analyse different aspects like the developers' role. For the sake of evaluation and replication, we openly published this database [21].

## D. Validation

In order to assess the relevance of our analysis, we validated the performance of our data extraction. In particular, we validated the accuracy of PAPRIKA and SNIFFER.

*1) PAPRIKA:* Hecht *et al.* [26] have already validated PAPRIKA with a F1-score of 0.9 in previous studies. They validated the accuracy of the used code smell definitions and the performance of the detection. The objective of our validation of PAPRIKA is to check that its detection is also accurate on our dataset. For this purpose, we randomly selected a sample of 599 code smell instances. We used a stratified sample to make sure to consider a statistically significant sample for each code smell. This represents a 95 % statistically significant stratified sample with a 10 % confidence interval of the 180, 013 code smell instances detected in our dataset. The stratum of the sample is represented by the 8 studied code smells. After the selection, one author manually analysed the instances to check their correctness. We found that all the sample instances are conform to the adopted definitions of code smells. Hence, we can affirm that the PAPRIKA code smell detection is effective in our dataset. The validated sample can be found with our artifacts [20].

*2) SNIFFER:* We aimed to validate the accuracy of the code smell history generated by SNIFFER. For this, we randomly selected a sample of 384 commits from our dataset. This represents a 95 % statistically significant stratified sample with a 5 % confidence interval of the 255, 798 commits in our dataset. After the selection, one author analysed every commit to check that the detected code smell introductions and removals are correct, and the SNIFFER did not miss any code smell introductions and removals. The results of this analysis can be found with our artifacts [20]. Based on these results, we computed the numbers of *true positives* (TP), *false positives* (FP), and *false negatives* (FN). These numbers are reported in Table II.

We did not find any case of missed code smell introductions or removals, $FN = 0$. However, we found cases where false code smell introductions and removals are detected, $FP = 7$ for both of them. These false positives are all due to a commit

TABLE II
VALIDATION OF SNIFFER.

| | ***TP*** | ***FP*** | ***FN*** | ***Precision*** | ***Recall*** | ***F1-score*** |
|---|---|---|---|---|---|---|
| ***Introductions*** | 151 | 7 | 0 | 0.95 | 1 | 0.97 |
| ***Removals*** | 85 | 7 | 0 | 0.92 | 1 | 0.96 |

from the Shopping List app [1], which renamed 12 Java files. Three of these renamings were accompanied with major modifications in the source code. Thus, the similarity between the files was above 50 % and Git could not detect the renaming. Consequently, SNIFFER could not track the code smells of these files and detected 7 false code smell introductions and removals.
Using the results of the manual analysis, we computed the precision, recall, and F1-score. Their values are reported in Table II. According to these measures, we can affirm that SNIFFER is effective for detecting both code smell introductions and removals.

## E. Data Analysis

Table III reports a list of metrics that we defined for the data analysis. We defined the rate metrics, namely

TABLE III
DEVELOPER METRICS.

| ***Metric*** | ***Description*** |
|---|---|
| *#commits* | (int): the number of authored commits. |
| *#introductions* | (int): the number of code smells introduced. |
| *#removals* | (int): the number of code smells removed. |
| *%introductions* | (float): introduction rate, *i.e.*, the average number of smells introduced per commit: $\#introductions/\#commits$. |
| *%removals* | (float): removal rate, *i.e.*, the average number of smells removed per commit: $\#removals/\#commits$. |
| *%contribution* | (float): the size of the developer's contribution compared to the project size: $\#commits/\#project\ size$. |
| *#followers* | (int): the number of followers. |
| *introducer* | (boolean): true if the developer introduced at least one code smell. |
| *remover* | (boolean): true if the developer removed at least one code smell. |
| *neutral* | (boolean): true if the developer did not introduce or remove any code smell. |
| *newcomer* | (boolean): true if the developer has less than three commits in the project. |
| *unfamous* | (boolean): true if the developer has less followers than 75% of the population. |
| *famous* | (boolean): true if the developer has more followers than 75% of the population. |

`%introductions` and `%removals`, to compare the developers' role in terms of code smells regardless of the size of their contributions. Indeed, comparing the number of code smells introduced by a developer who has 100 commits and another who has only 10 commits can be biased. The developer with more commits has more chances to introduce and remove code smells. Similarly, the metric `%contribution` allows to relate the number of commits to the project size. For instance, a contribution of 10 commits in a project of 20 commits is totally different from a contribution of 10 commits in a project of 100 commits. For the definition of the metrics `newcomer` and `regular`, we followed the approaches proposed in previous studies. As a matter of fact, Tufano *et al.* [49] used the threshold of 3 commits to distinguish newcomers from regular developers. We present in the following how we used these metrics in order to answer our research questions.

### *1) RQ1: How do developers contribute in code smell introductions and removals?:*

The analysis of this research question consisted in testing its three hypotheses.

***H1**: Code smell introductions and removals are isolated practices.*
This hypothesis suggests that only an isolated part of Android developers are responsible for code smell introductions and removals. To test this hypothesis, we investigated the distribution of developers regarding code smell introductions and removals. Specifically, we computed the numbers and percentages of developers that have the attributes `introducer`, `remover`, and `neutral`. This allowed us to split the developers into different groups and have an insight about the general tendency of code smell introductions and removals among Android developers.

***H2**: There is an inverse correlation between code smell introductions and removals among developers.*
We took a two-step approach to test this hypothesis. First, we measured the relationship between the metrics `#introductions` and `#removals` using Spearman's rank correlation 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.
As the metrics `#introductions` and `#removals` can be biased by the number of commits, we proceeded to the second step with different metrics. Specifically, we used Spearman to measure the correlation between `%introductions` and `%removals`.

***H3**: For each code smell type, there is an inverse correlation between code smell introductions and removals among developers.*
This hypothesis states that at the code smell level, the introduction and removal are inversely correlated. This means that developers who are unaware about a specific type of code smells, will tend to introduce it and never remove it. Similarly, developers who are aware about the code smell type will tend to remove it and never introduce it. To test this hypothesis, we computed the metrics `%introductions` and `%removals` for each type of code smell, *e.g.*, `%introductions_LIC`, `%removals_LIC` for the *LIC* code smell. As we study 8 code smells, this process gave us 16 metrics. Afterwards, we computed the correlation between the two metrics of each code smell type, *e.g.*, $correlation(\%introductions\_MIM, \%removals\_MIM)$.
The correlation coefficient used is again Spearman's rank.

*2) RQ2: How does seniority impact developers contribution?:* The objective of this research question is to assess the impact of the developer's seniority on code smell introductions and removals. We measured the seniority with (1) the experience in the development project and (2) the reputation in the community.

***H4**: Newcomers tend to introduce more and remove less code smells.*
To assess this hypothesis, we followed two approaches. First, we split the developers into two groups using the metrics `newcomer` and `regular`. Then, we compared the distribution of the metrics `%introductions` and `%removals` in the two groups. In particular, we used a two-tailed Mann-Whitney U test [46], with a 99 % confidence level, to check if the distributions of introductions and removals are identical in the two sets. To quantify the effect size of the presumed difference, we used Cliff's $\delta$ [45]. Cliff is a non-parametric effect size measure which is reported to be more robust and reliable than Cohen's d [11]. Moreover, it is suitable for ordinal data and it makes no assumptions of a particular distribution [45]. 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 $|d| \geq 0.474$.

In the second approach, we relied on the metric `%contribution` which reflects the involvement of a developer in a project. We assessed the impact of this involvement on code smell introductions and removals. Concretely, we computed the Spearman's rank of the metric `%contribution` with `%introductions` and `%removals` respectively.

***H5**: Reputable developers tend to introduce less and remove more code smells.*
In this hypothesis, we opted for the number of followers as an estimation of the developer's reputation. The number of followers is a signal of status in the community, developers with many followers are treated as celebrities [14].

To assess the hypothesis, we first used the metrics `famous` and `unfamous` to split the developers into two groups. Then, we compared the tendency of code smell introductions and removals in the two groups using the metrics `%introductions` and `%removals`. To perform this comparison we relied on the quartiles, Mann-Whitney U, and Cliff's $\delta$.

As a second approach, we measured directly the correlation between the reputation and code smell introductions and removals. For this, we computed the Spearman's rank of the metric `#followers` with `%introductions` and `%removals`, respectively.

## III. Study Results

In this section, we report and analyse the results of our experiments with the aim of answering our research questions.

### *A. RQ1: How do developers contribute in code smell introductions and removals?*

*1) H1:* Table IV reports the distribution of developers in terms of code smell introductions and removals. We observe that only 35 % of the developers are implicated in code smell introductions. The remaining 65 % did not introduce any code smell instances. The distribution in terms of code smell removals is similar. Only 31 % of the developers participated in code smell removals. We also observe that 28 % of the developers are implicated in both code smell introductions and removals. This intersection between introducers and removers is highlighted in Figure 3. The figure shows that there is an important intersection between developers that introduce and remove code smells. This means that developers who introduce and remove code smells are mainly the same. Another interesting observation from Table IV is that 61 % of developers are neutral, they did not introduce or remove any code smell instances.

TABLE IV
Distribution of developers according to code smell introductions and removals.

| | *introducer* | *remover* | *introducer and remover* | *neutral* | *All* |
|---|---|---|---|---|---|
| *# developers* | 1590 | 1414 | 1269 | 2790 | 4525 |
| *% developers* | 35 | 31 | 28 | 61 | 100 |

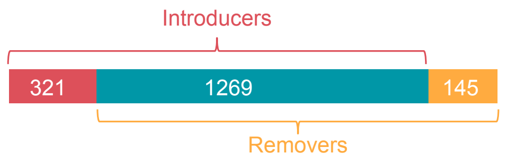

Fig. 3. Intersection between introducers and removers.

To further investigate the nature of these groups, we compare in Figure 4 the distribution of their number of commits. The distribution is illustrated with a density function, which allows to highlight how many commits are authored by the developers in the three groups.

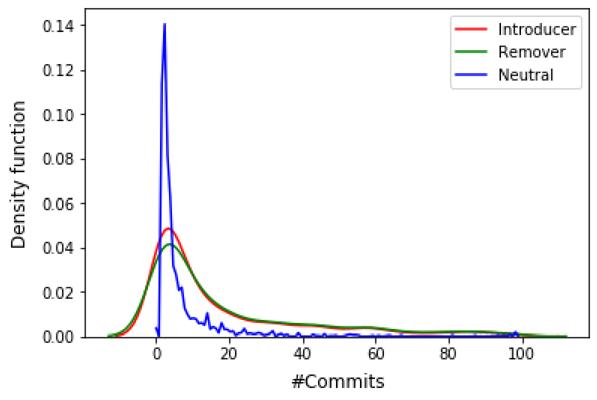

Fig. 4. The distribution of the number of commits among the studied developers.

The first thing that leaps to the eye is that neutral developers are concentrated around a low number of commits. Most of them have only authored less than 5 commits. This means that developers who did not introduce or remove any code smells are developers who did not commit regularly. On the other side, introducers and removers are less concentrated around the low values. Developers in these two groups have authored more commits. It is also worth noting that introducer and remover groups are similarly distributed which aligns with Figure 3 where they manifested a considerable intersection.

To sum up, even if Table IV shows that introducers and removers are only one third of the total contributors. They actually represent most of the regular developers. Developers who did not participate in code smell introductions and removals have very few commits. Based on these findings, we can assert that code smell introduction and removal are not isolated practices. Thus, we can reject Hypothesis 1.

Developers who introduce and remove code smells are mainly the same. Developers who did not introduce or remove code smells are not regular contributors.

*2) H2:* The results of the correlation between the numbers of code smell introductions and removals among developers are the following.

$$Spearman(\#introdcutions, \#removals) \begin{cases} r = 0.94 \\ p\text{-}value < 0.01 \end{cases}$$

The $p - value$ shows that the computed $r$ coefficient is statistically significant. The coefficient value, 0.94, shows that there is a strong positive relationship between the numbers of introductions and removals performed by a developer. That is, the more a developer tends to introduce code smells, the more she will remove. To further investigate this finding, we evaluated the relationship with other measures. The results of the correlation between the introduction and removal rates are the following.

$$Spearman(\%introdcutions, \%removals) \begin{cases} r = 0.77 \\ p\text{-}value < 0.01 \end{cases}$$

As the $p - value$ is lower than the threshold 0.01, the computed coefficient is statistically significant. The coefficient value, 0.77, shows that the introduction and removal rates are positively correlated. This means that the more a developer introduces code smells per commit, the more she removes as well. Hence, even independently of the number of commits, code smell introduction and removal are positively correlated among developers. Based on these results, we can reject Hypothesis 2.

The more a developer introduces code smells, the more she tends to remove them.

*3) H3:* Table V reports the correlation coefficients between the `%introductions` and `%removals` per code smell type. For the eight code smells, the $p - values$ are under the threshold, thus the results are statistically significant. The correlation coefficients depict that the introduction and removal rates are positively correlated. Indeed, for *LIC*, *MIM*, *NLMR*, *UHA* and *UCS*, the introduction and removal rates are strongly correlated with a coefficient over 0.8. This means that developers who introduced these types of code smells also removed them and vice versa. For *HMU*, *UIO*, and *IOD*, the correlation is between 0.5 and 0.6. This means that there is an average positive correlation between the introduction and removal rates of these code smells among developers. These findings show that developers remove the same type of code smells that they introduce. There is no code smell where developers who removed instances did not tend to introduce new ones. Hence, we can affirm that there is no inverse correlation between introductions and removals at the code smell level. Thus, we can reject Hypothesis 3.

TABLE V
CORRELATION BETWEEN INTRODUCTIONS AND REMOVALS PER CODE SMELL TYPE.

| *Smell* | *LIC* | *MIM* | *NLMR* | *HMU* | *UIO* | *UHA* | *IOD* | *UCS* |
|---|---|---|---|---|---|---|---|---|
| *Correlation* | 0.9 | 1.0 | 0.8 | 0.6 | 0.6 | 0.8 | 0.5 | 1.0 |
| *p-value* | $< 0.01$ | | | | | | | |

The more a developer introduces a type of code smells, the more she removes it.

## B. RQ2: How does seniority impact developers contribution?

*1) H4:* Table VI compares the introduction and removal rates among newcomers and regular contributors. First, we observe that using the criterion defined in the study design, our dataset contains 3375 newcomers and 1793 regular contributors. It is worth noting that the sum of these two groups is above the total number of developers 4,525. This due to the involvement of many of developers in several studied projects. Thus, these developers did not have one global contribution in this context, but multiple contributions.

TABLE VI
COMPARISON BETWEEN NEWCOMERS AND REGULAR DEVELOPERS IN TERMS OF INTRODUCTION AND REMOVAL RATES.

| | | *size* | *Q1* | *Med* | *Q3* | *p-value* | *Cliff* |
|---|---|---|---|---|---|---|---|
| *%Introduction* | *Newcomer* | 3375 | 0 | 0 | 0 | $< 0.01$ | $0.46(M) \downarrow$ |
| | *Regular* | 1793 | 0 | 0.21 | 0.76 | | |
| *%Removal* | *Newcomer* | 3375 | 0 | 0 | 0 | $< 0.01$ | $0.44(M) \downarrow$ |
| | *Regular* | 1793 | 0 | 0.11 | 0.50 | | |

Regarding introductions, we observe that newcomers tend to introduce very few code smells per commit, *Q1, Median,* and *Q3* are null. This means that at least 75 % of the new comers did not introduce any code smell per commit. On the other side, regular contributors tend to introduce more code smells. Indeed, 50 % of them introduce more than 0.21 code smells per commit and 25 % of them are above 0.76 introductions. This superiority is confirmed by the significant *p-value* computed by Mann-Whitney U and the Cliff's $\delta$ value 0.46. The latter shows that there is a medium difference in the introduction rates in favour of regular contributors.

As for removals, the table shows that newcomers do not tend to remove code smells. *Q1, Median,* and *Q3* values are null. Regular contributors tend to remove more code smells. 50 %

of them remove more than $0.11$ code smells per commit and $75\,\%$ of them have a removal rate above $0.50$. Mann-Whitney U and Cliff's $\delta$ values confirm this observation. Removal rates among regular contributors are mediumly superior to the rates of newcomers.

To push forward this analysis, we computed the correlation between the experience in the project and code smell introductions and removals. The results are the following.

$$Spearman(\%contribution, \%introductions) \begin{cases} r = 0.27 \\ p\text{-}value < 0.01 \end{cases}$$

$$Spearman(\%contribution, \%removals) \begin{cases} r = 0.24 \\ p\text{-}value < 0.01 \end{cases}$$

In both correlations, the $p - value$ indicates that the results are statistically significant. For the introduction, the $r$ value, $0.27$ shows that there is a weak positive correlation between the experience in the project and the introduction rate. Likewise for code smell removals, the $r$ value, $0.24$, indicates that the correlation between experience and removals is positive and weak. This means that experience does not have an impact on the developer's role in terms of code smells. Developers with few contributions, like newcomers, do not forcibly introduce more or remove less code smells. Hence, based on the comparison and correlation results, we can reject Hypothesis 4.

Newcomers are not particularly bad contributors. They do not introduce or remove more code smells than regular developers.

*2) H5:* Figure 5 and Table VII show the distribution of the number of followers among the studied developers.

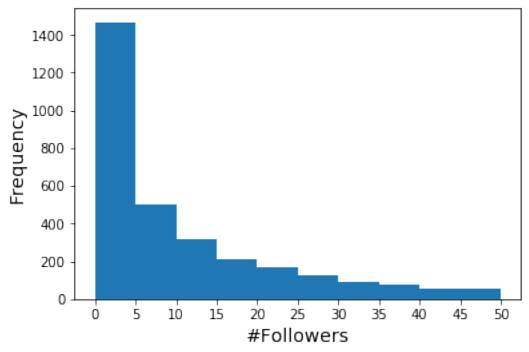

Fig. 5. The distribution of the number of followers among studied developers.

TABLE VII
THE NUMBERS OF FOLLOWERS AMONG THE STUDIED DEVELOPERS.

| | *Q1* | *Median* | *Q3* | *IQR* |
|---|---|---|---|---|
| ***#Followers*** | 1 | 8 | 25 | 24 |

We observe that the majority of the studied developers have less than $25$ followers. Based on the values of Table VII, we defined our thresholds for splitting developers into two groups:

- `unfamous`: developers with $\leq 1$ followers,
- `famous`: developers with $\geq 25$ followers.

Table VIII compares the code smell introduction and removal rates among unfamous and famous developers.

TABLE VIII
COMPARISON BETWEEN UNFAMOUS AND FAMOUS DEVELOPERS IN TERM OF CODE SMELL INTRODUCTION AND REMOVAL RATES.

| | | *size* | *Q1* | *Med* | *Q3* | *p-value* | *Cliff* |
|---|---|---|---|---|---|---|---|
| ***%Introduction*** | ***Unfamous*** | 945 | 0 | 0 | 0 | $< 0.01$ | $0.18(S) \downarrow$ |
| | ***Famous*** | 906 | 0 | 0 | 0.47 | | |
| ***%Removal*** | ***Unfamous*** | 945 | 0 | 0 | 0 | $< 0.01$ | $0.19(S) \downarrow$ |
| | ***Famous*** | 906 | 0 | 0 | 0.29 | | |

For the introduction rate, we observe that in both groups, the introduction per commit is around zero for most of the developers. Indeed, *Q1* and *Median* values are null for both unfamous and famous developers. The difference arises in the third quartile values, *Q3*. The latter shows that whereas $25\%$ of famous developers introduce more than $0.47$ code smells per commit, unfamous developers are mostly around $0$ introductions per commit. This difference is confirmed by the Mann-Whitney U results as the *p-value* is below the confidence threshold. The Cliff's $\delta$ value, $0.18$, shows that there is a small difference between the two groups in favour of famous developers. This means that famous developers have a slightly higher introduction rate than the unfamous ones.

We observe similar results for removal rates. For both groups, $50\,\%$ of the developers remove $0$ code smells per commit—*Median=0*. However, the *Q3* value shows that $25\,\%$ of famous developers introduce more than $0.29$ code smells per commit while the unfamous ones are mostly around $0$ introductions. Mann-Whitney U and Cliff's $\delta$ results confirm this observation. The *p-value* is below the threshold which means that there is a statistically significant difference between the removal rates in both groups. The Cliff's $\delta$ shows that, while small $0.19$, this difference is in favour of famous developers. This means that famous developers tend to remove slightly more code smells than unfamous ones.

To consolidate this analysis, we computed the correlation between the developer's reputation and her code smell introductions and removals. The results are the following.

$$Spearman(\#followers, \%introductions) \begin{cases} r = 0.14 \\ p\text{-}value < 0.01 \end{cases}$$

$$Spearman(\#followers, \%removals) \begin{cases} r = 0.15 \\ p\text{-}value < 0.01 \end{cases}$$

As the $p - values$ are below the threshold, we can consider the computed coefficients as statistically significant. The correlation coefficient between the number of followers and the introduction rate, $0.14$, suggests that there is a very weak positive correlation between them. Likewise, the correlation between the number of followers and the removal rates is weak, $r = 0.15$. This means that the reputation does not have an impact on the code smell activity. That is, famous developers are not significantly better contributors in terms of code smells. Based on these findings, we can reject Hypothesis 5.

Reputable developers are not particularly good contributors in terms of code smells.

## IV. Discussion & Threats

### A. Discussion

We conducted this study with the hypothesis that some developers are responsible for the accrual of mobile code smells. In light of our results, we were able to reject this hypothesis [18], [19], [26]. Even if the numbers show that $61\,\%$ of the developers are neutral, this percentage is biased by the huge number of developers who had very few commits. Indeed, developers who committed regularly are more present and had more chances to introduce or remove code smells. This means that the phenomenon of code smells is not limited to a small group of developers. The results of *RQ2* pushed this idea forward. Besides not being a small isolated group, developers who contributed to code smells are not necessarily newcomers or unfamous in their communities. This implies that there is no isolated group of junior developers to blame for code smells, the responsibility is rather shared among all regular contributors. Our analysis also showed that developers who introduce and remove code smells are not distinct, they are mainly the same group. On top of that, among these developers, the introduction and removal tendencies are correlated, even at the smell level. This implies that code smell removals are not intended. Indeed, a developer who consciously removes a code smell, would not tend to introduce it elsewhere. This behaviour appears more like accidental introductions and removals of code smells. Therefore, we challenge the suggestion that Android developers *"perform refactorings to improve the code structure and enforce solid programming principles"* [29]. In fact, similar observations were previously made about the refactoring of OO code smells. Studies showed that sometimes code smells refactoring is due to maintenance activities and cannot be considered as a proper refactoring [41], [49].

Based on this analysis, we suggest that the studied mobile code smells represent an inadvertent technical debt—*i.e.*, a debt which is driven by oversight instead of strategic or tactic choices [48]. That is, developers do not deliberately accept this technical debt in favour of other objectives, like productivity. Rather, they introduce the debt with no leverage—*i.e.*, they do not gain anything in return. To refine and further explore this hypothesis, we encourage future research works to:

- Analyse the nature of Android code smells removals and check whether it is a proper refactoring or a side effect of source code evolution activities.
- Investigate the role of other technical debt drivers, like prioritisation and releases [48], in the accrual of Android code smells.
- Study the self-admittance [13] of Android code smells.

The observed correlations also show, with quantitative evidence, that the unawareness described by developers in the study of Habchi *et al.* [18] is very prevalent. This oversight incites us to question the importance of mobile code smells and the efficiency of the communication channels used by framework builders.
Effectively, the unawareness of most developers may be due to the unimportance of these code smells in practice. This questioning is particularly relevant as the code smells are performance-oriented. If these code smells effectively impacted apps in practice, the performance degradation would have attracted developers awareness. This hypothesis has the following implications on researchers:

- Studies should reevaluate the importance and impact of mobile code smells in practice. We highly encourage future works to perform empirical studies on real life scenarios and real mobile apps to assess the effective impact of these code smells on performance.
- The definition of code smells should not only be a top-down process. That is, mobile code smells, which are for now only based on documentation, should also consider the viewpoint of developers. We invite researchers to involve developers in the evaluation of these code smells and the definition of future ones. The developers' opinion would allow us to better evaluate the importance and severity of these code smells in practice.

The other possible explanation of the prevalent unawareness towards mobile code smells is the vague communication about them. It is unlikely that all developers carelessly consider the framework documentation, especially when it comes to performance. Hence, we suggest for framework builders to:

- Highlight and stress the importance and impact of code smells.
- Use more pervasive channels to raise developers awareness about code smells. In particular, framework built-in tools, like linters, should emphasise the importance and impact of code smells.
- Researchers and tool makers should work on just-in-time detection and refactoring tools that run as soon as the change is committed to prevent the accumulation of code smells.

### B. Threats to Validity

*Internal Validity:* : For this study, one main major threat could be to associate code smell introductions or removals to the wrong developers. This can occur in situations where the code smell is introduced or removed gradually, making it complex to determine the introducing or removing commit. However, as explained previously, we consider in this study only code smells that can only be introduced or removed in one commit. Another case that may cause confusion for smell introductions is merge commits. A merge commit groups many commits that may introduce and remove code smells. Sniffer solves this problem by using a branch-aware commit order [17]. Therefore, we can always consider that code smell introductions and removals are associated to the appropriate commit.

*External Validity:* : The main threat to external validity is the representativeness of our dataset. We used a set of $324$ open-source Android apps from F-Droid with more than $255k$ commits. 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 also encourage future studies to consider other datasets of open-source apps to extend this study [16], [33]. Another possible threat is that our study only concerns 8 Android-specific code smells. Without a closer inspection, these results should not be generalised to other code smells or mobile platforms. We encourage future studies to replicate our work on other datasets and with different code smells and mobile platforms.

*Construct Validity:* : In our case, the construct might be affected by the code smell detection performed by PAPRIKA. However, PAPRIKA has been validated with a precision of (0.88) and a recall of (0.93) [28]. On top of that, our validation showed that PAPRIKA is also effective in the dataset under study. Thus, we can rely on its performance.

*Conclusion Validity:* : The main threat to the conclusion validity in this study is the validity of the statistical tests applied. We alleviated this threat by applying a set of commonly accepted tests employed in the empirical software engineering community [35]. Moreover, we used non-parametric tests that do not require making assumptions about the distribution of the data.

## V. RELATED WORKS

*Mobile code smells:* The first reference to mobile-specific code smells was when Reimann *et al.* [42] proposed a catalog of 30 quality smells dedicated to Android. Later, researchers proposed tools and approaches like PAPRIKA [28] and ADOCTOR [38] to detect these code smells. Relying on these tools, empirical studies were conducted to prove the negative impact of mobile-specific code smells on app performance [9], [27], [39]. Other empirical studies focused on understanding the phenomenon of code smells in mobile apps. In particular, Mannan *et al.* [34] compared the presence of OO code smells in Android and desktop applications. They observed that the distribution of code smells in Android is more diversified than in desktop applications. Habchi *et al.* [19] also opted for a comparison between the Android and iOS specific code smells. They observed semantic similarities between the code smells exhibited by the two platforms. Interestingly, none of these studies studied code smells from the developers angle. They only approached this topic in their explanatory hypotheses when discussing the possible factors of mobile code smells accumulation. Our study builds on these hypotheses and addresses the developers role with a large scale analysis.

*Developers and code smells:* The closest work to our study is the one of Habchi *et al.* [18] where they showed that some Android developers are unaware of the importance and impact of performance bad practices in mobile apps. However their findings were based on a qualitative study and they did not provide any quantitative assessments of the prevalence of this unawareness. Regarding OO code smells, Peters *et al.* [41] conducted a case study on 7 open-source systems and investigated the refactoring behavior of developers. They observed that usually one or two developers refactor more than the others, however the difference is not large. Later, Tufano *et al.* [49] analysed the change history of 200 open-source projects to understand why OO code smells are introduced. Among other findings, they observed that newcomers are not necessarily more prone to introducing new code smells.

*Mining the change history:* Mining the change history is used for various purposes in software engineering. Studies particularly relied on it for topics like defect prediction and code ownership analysis [7], [24], [25], [47]. However, researchers observed that Git, the *de facto* VCS platform, is not mining-friendly [6], [23], [30]. Indeed, Git was not designed for an accurate retrieval of history changes [32]. For this purpose, many studies highlighted Git pitfalls and urged the MSR community to address them [6], [23], [31]. In particular, a recent study of Kovalenko *et al.* [32] showed that handling branches and renamings is crucial for an accurate tracking of contributions in Git. Our proposed toolkit, SNIFFER, complies with these guidelines and proposes a renaming and branch aware analysis of the change history.

## VI. CONCLUSION

We presented in this paper the first large-scale empirical study about the role of developers in the accrual of mobile-specific code smells. We analysed 8 Android code smells, 324 Android apps, $255k$ commits, and contributions from $4,525$ developers. This study resulted in several findings that challenge the conventional wisdom:

**Finding 1:** The accrual of Android code smells is not the responsibility of an isolated group of developers. Most regular developers participated in the introduction of code smells.

**Finding 2:** There are no distinct groups of code smell *introducers* and *removers*. Developers who introduce and remove code smells are mostly the same.

**Finding 3:** While newcomers are not to blame for the accrual of mobile code smells, reputable developers are not necessarily good contributors.

These findings highlight important challenges for researchers and framework builders. In particular, researchers should reevaluate the importance and impact of mobile code smells, and consider involving developers in the definition of future code smells. Also, framework builders should put more efforts in creating an actionable documentation and pervasive tools to raise developers awareness.

On top of the research implications, this paper proposes SNIFFER, a novel open-source toolkit [22] that mines the change history of Android apps to track developers' contributions. SNIFFER tackles many issues raised by the Git mining community like branch and renaming tracking. We also provide all the collected data for this paper in a database that contains the history of 180.013 Android code smells [21]. We highly encourage the community to build on our findings, toolkit, and dataset to replicate and perform further studies on developers and mobile code smells.

## References

[1] Commit from shopping list app. https://github.com/openintents/shoppinglist/commit/efa2d0b2214349c33096d1d999663585733ec7b7#diff-7004284a32fa052399e3590844bc917f, 2014. [Online; accessed December-2018].

[2] Git diff core delta algorithm. https://github.com/git/git/blob/6867272d5b5615bd74ec97bf35b4c4a8d9fe3a51/diffcore-delta.c, 2016. [Online; accessed November-2018].

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

[4] Android. Android Lint - Android Tools Project Site. http://tools.android.com/tips/lint-checks, 2017. [Online; accessed July-2018].

[5] Android. Deprecation of invalidate with rect. https://developer.android.com/reference/android/view/View.html\#invalidate(), 2017. [Online; accessed January-2019].

[6] V. Arnaoudova, L. M. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y.-G. Gueheneuc. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, 40(5):502–532, 2014.

[7] R. M. Bell, T. J. Ostrand, and E. J. Weyuker. The limited impact of individual developer data on software defect prediction. *Empirical Software Engineering*, 18(3):478–505, 2013.

[8] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52. ACM, 2010.

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

[10] Cast. Reduce technical debt. https://www.castsoftware.com/research-labs/technical-debt, 2017. [Online; accessed January-2019].

[11] J. Cohen. A power primer. *Psychological bulletin*, 112(1):155, 1992.

[12] W. Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1993.

[13] E. da Silva Maldonado, E. Shihab, and N. Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, 2017.

[14] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM, 2012.

[15] J. ELM. Design debt economics: A vocabulary for describing the causes, costs and cures for software maintainability problems. ibm, 2009.

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

[17] Git. Git Commit oredering. https://git-scm.com/docs/git-log, 2017. [Online; accessed August-2017].

[18] S. Habchi, X. Blanc, and R. Rouvoy. On adopting linters to deal with performance concerns in android apps. In *ASE18-Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*, volume 11. ACM Press, 2018.

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

[20] S. Habchi, N. Moha, and R. Rouvoy. Companion artifacts. https://figshare.com/s/26d66505a75a772dd994, 2019. [Online; accessed January-2019].

[21] S. Habchi, N. Moha, and R. Rouvoy. Database of full code smell history. https://figshare.com/s/ffa0a64a55e040f48dc9, 2019. [Online; accessed January-2019].

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

[23] A. E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE, 2008.

[24] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[25] L. Hattori and M. Lanza. Mining the history of synchronous changes to refine code ownership. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 141–150. IEEE, 2009.

[26] G. Hecht. *Détection et analyse de l'impact des défauts de code dans les applications mobiles*. PhD thesis, Université du Québec à Montréal, Université de Lille, INRIA, 2017.

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

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

[29] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien. Detecting Antipatterns in Android Apps. Research Report RR-8693, INRIA Lille ; INRIA, Mar. 2015.

[30] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, 19(2):77–131, 2007.

[31] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 58–64. ACM, 2006.

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

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

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

[35] K. Maxwell. *Applied statistics for software managers.* Prentice Hall, 2002.

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

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

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

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

[40] R. Pawlak. Spoon: Compile-time annotation processing for middleware. *IEEE Distributed Systems Online*, 7(11), 2006.

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

[42] J. Reimann, M. Brylski, and U. Aßmann. A tool-supported quality smell catalogue for android developers. In *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM*, volume 2014, 2014.

[43] J. Reimann, M. Brylski, and U. Aßmann. A Tool-Supported Quality Smell Catalogue For Android Developers. In *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung – MMSM 2014*, 2014.

[44] J. Reimann, M. Seifert, and U. Aßmann. On the reuse and recommendation of model refactoring specifications. *Software & Systems Modeling*, 12(3):579–596, 2013.

[45] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek. 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*, pages 1–33, 2006.

[46] D. J. Sheskin. *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2003.

[47] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering*, pages 1039–1050. ACM, 2016.

[48] E. Tom, A. Aurum, and R. Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013.

[49] M. Tufano, F. Palomba, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, PP, 2017.