# On the Survival of Android Code Smells in the Wild

Sarra Habchi, Romain Rouvoy, Naouel Moha

# On the Survival of Android Code Smells in the Wild

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

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

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

***Abstract*—The success of smartphones and app stores have contributed to the explosion of the number of mobile apps proposed to end-users. In this very competitive market, developers are rushed to regularly release new versions of their apps in order to retain users. Under such pressure, app developers may be tempted to adopt bad design or implementation choices, leading to the introduction of code smells. Mobile-specific code smells represent a real concern in mobile software engineering. Many studies have proposed tools to automatically detect their presence and quantify their impact on performance. However, there remains—so far—no evidence about the lifespan of these code smells in the history of mobile apps. In this paper, we present the first large-scale empirical study that investigates the survival of Android code smells. This study covers $8$ types of Android code smells, $324$ Android apps, $255k$ commits, and the history of $180k$ code smell instances. Our study reports 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. Also, Android code smells disappear faster in bigger projects with higher releasing trends. Finally, we observed that code smells that are detected and prioritised by linters tend to disappear before other code smells.**

***Index Terms*—Mobile apps, Android, code smells.**

## I. Introduction

Code smells are well-known in *Object-Oriented* (OO) software systems as poor or bad practices that negatively impact the software maintainability and cause long-term problems. Since their introduction by Fowler [15], the research community has shown an increasing interest in code smells. Many studies investigated and quantified their presence and impact on source code [28], [39]. Other studies focused on the evolution of code smells in the change history (project commits). In particular, Peters and Zaidman [46] showed that code smell instances have a lifespan of approximately 50 % of the revision history. Also, Tufano *et al.* [53] demonstrated that $50\%$ of code smells instances persisted in the codebase for more than $1,000$ days and $1,000$ commits from their introduction. These studies helped in improving our understanding of code smells and highlighted notable facts, like the lack of awareness from developers about code smells.

Nowadays, mobile applications (apps) retain the highest share in software market, and they differ significantly from traditional software systems [38]. In fact, the development of mobile apps must take into consideration device limitations like memory, CPU, and energy. These limitations motivated the identification of mobile-specific code smells [48]. Mobile code smells are different from OO code smells as they often refer to a misuse of the platform SDK and they are more performance-oriented. This performance aspect drew the attention of the research community as many studies investigated the impact of these code smells on performances, demonstrating that their refactoring can significantly improve app performance [11], [23], [44]. Apart from performance, other aspects of mobile-specific code smells remained unaddressed. Specifically, we still lack knowledge about the lifespan of mobile code smells in the apps history. This knowledge is particularly important for understanding the extent and importance of these code smells. Effectively, a code smell that persists in the software history for long periods can be more critical than code smells that disappear in only a few commits. Moreover, the survival analysis of these code smells allows us to understand the factors that favour or prevent their expansion in source code. Understanding such factors is crucial for proposing solutions that handle mobile code smells. For these reasons, we investigate in this paper the survival of Android-specific code smells. More specifically, we answer the following research questions:

- **RQ1:** For how long do Android-specific code smells survive in the codebase?
- **RQ2:** What are the factors that impact the survival of Android-specific code smells? We investigate in this question the impact of project size, releasing practices, and code smell properties on the survival chances of code smell instances. The effect of the project size interests us as the common sense suggests that code smells persist longer in big and complex projects. Regarding releases, Android apps are known for having more frequent releases and updates [37]. Our aim is to investigate the impact of this particularity on code smell survival. As for code smell properties, we are particularly interested in the impact of Android Lint [5]. Android Lint is the mainstream linter—*i.e.*, static analyser, for Android. It is integrated and activated by default in the official IDE Android Studio. Our objective is to inspect whether the code smells detected and prioritised by Android Lint are removed faster from the codebase.

This paper has the following notable contributions:

1) The first large-scale empirical study that investigates the evolution of mobile-specific code smells in the change history. This study covers $8$ types of Android code smells, $324$ Android

apps, $255k$ commits, and the history of $180k$ code smells.

2) Among other findings, our study shows that, while in terms of time Android code smells can persist in the codebase for years, it only takes 34 effective commits to remove $75\%$ of them. Our results also show that Android code smells disappear faster in bigger projects with more commits, developers, classes, and releases. Finally, code smells that are detected and prioritised by Android Lint tend to disappear before other code smell types.

The remainder of this paper is organised as follows. Section II explains the study design and Section III reports on the results. Section IV discusses the implications and the threats to validity, while Section V analyses related works. Finally, Section VI concludes with our main findings and perspectives.

## II. Study Design

We start this section with a presentation of our study context. Afterwards, we explain our data extraction technique. 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 the analysis of mobile apps history to investigate the survival of mobile-specific 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 [18]. On top of that, both development and research communities proposed tools to analyse the source code of Android apps and detect their code smells [5], [25], [43].

*2) Code Smells & Detection Tool:* In the academic literature, the main reference to Android code smells is the catalog of Reimann *et al.* [47]. 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], [6]. 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 [25]. We present in Table I brief descriptions of the eight code smells that we kept for our study.

### B. Dataset and 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 are available on GitHub. We automatically crawled the apps available on FDroid and retrieved their GitHub links when available. Then using these links, we fetched the repositories from GitHub. For computational constraints, we only kept repositories which had at least two developers. This filter resulted in 324 projects with $255,798$ commits. The full list of projects can be found in our companion artifacts [19].

### C. Data Extraction

We performed our data extraction using Sniffer [20], an open-source toolkit that tracks the full history of Android-specific code smells. Figure 1 depicts an overview of the Sniffer process. A full description of this process is available in our technical report and the source code is openly published [19]. In this paper we only explain briefly the main steps of the approach and how we use it in the context of our study.

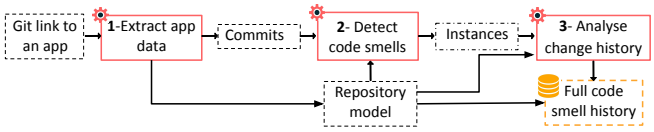


Fig. 1: Overview of the Sniffer toolkit.

*1) Step 1: Extract App Data:*
**Input:** Git link to an Android app.
**Output:** Commits and repository model.
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, etc. One particular element that interests us in this study is the Git tag. Git tags are used to label specific points of the change history as important. Typically, developers use them to label release points [51]. We will rely on these tags to study the impact of releases on the code smell survival. Another important information extracted in this step is the branch property. 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.

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

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

| *Definition* | *Entity* |
|---|---|
| **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], [48]. | Inner class |
| **Member Ignoring Method (MIM)**: this smell occurs when a method, which 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 [24]. | Method |
| **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 [48]. | Activity |
| **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 *vs.* 4 bytes respectively. Therefore, the framework recommends using the `SparseArray` data structure, which is more memory-efficient [3], [48]. | Method |
| **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 [5], [48]. | View |
| **Unsupported Hardware Acceleration (UHA)**: in Android, most of the drawing operations are executed in the GPU. Drawing operations that are executed in the CPU (*e.g.*, `drawPath()` of `android.graphics.Canvas`) should be avoided to reduce CPU load [22], [41]. | View |
| **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]. | View |
| **Unsuited LRU Cache Size (UCS)**: this code smell occurs when an LRU cache 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 [22], [36]. | Method |

TABLE I: Studied code smells.

*2) Step 2: Detect Code Smells:*

**Input:** Commits and repository model.

**Output:** Code smell instances per commit.

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 a source code model. 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 [45], 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, properties, and metrics constitute a source code model that can be used 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.

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 retrieve the previous commits accurately and track renamings. Thanks to the branch information extracted in Step 1, SNIFFER is able to accurately retrieve the previous commits even in the cases of multiple branches and merge commits. As for renaming, SNIFFER relies on Git to tracks files with their contents instead of their names or paths. Git uses a similarity algorithm [2] to compute a similarity index between two files. By default, if the similarity index is below $50\,\%$, the two files are considered as the same. SNIFFER uses this feature to detect renamings and accurately track file and code smell history.

We used SNIFFER on the $324$ selected apps and extracted the full history of $180,013$ code smells. This history is saved in a POSTGRESQL database that can be queried for data analysis. For the sake of evaluation and replication, we openly published this database [19].

### D. Validation

In order to assess the relevance of our analysis, we made sure that the tools used in our study are validated on our dataset.

*1)* PAPRIKA*:* Hecht *et al.* [22] 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 [19].

*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 detailed results of this analysis can be found in our companion artifacts [19]. 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.

| | ***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 |

TABLE II: Validation of SNIFFER.

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 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*

We explain in this sub-section our data analysis approach for answering our two research questions.

*1) RQ1: For how long do Android-specific code smells survive in the codebase?:* To answer this research question, we relied on the statistical technique of survival analysis, *a.k.a.* time-to-event analysis [31]. The technique analyses the expected duration of time until one or more events happen (*e.g.*, a death in biological organisms or a failure in mechanical systems). This technique suits well our study since we are interested in the duration of time until a code smell is removed from the codebase. Hence, in the context of our study, the subjects are code smell instances and the event of interest is their removal from the codebase. As for the time-to-event, it refers to the lifetime between the instance introduction and its removal.

The survival function $S(t)$ is defined as:

$$S(t) = Probability(T > t)$$

Where $T$ is a random lifetime from the population under study. That is, the survival function defines the probability for a subject (a code smell in our case) for surviving past time $t$. The survival function has the following properties [14]:

1) $0 \leq S(t) \leq 1$;
2) $S(t)$ is a non-increasing function of $t$;
3) $S(0) = 1$ and $\lim_{t\to\infty} S(t) \to 0$.

Interestingly, survival analysis models take into account two data types:

1) **Complete data**: this represents subjects where the event of interest was already observed. In our study, this refers to code smells that were removed from the codebase.
2) **Censored data**: this represents subjects that left during the observation period and the event of interest was not observed for them. In our case, this refers to code smells that were not removed during the analysed commits.

To measure the lifetime of code smell instances, we relied on two metrics:

1) #Days: The number of days between the commit that introduces the code smell and the one that removes it.
2) #Effective commits: The number of commits between the code smell introduction and removal. For this metric, we only counted commits that performed modifications in the file of the code smell. This fine grained-analysis allows us to exclude irrelevant commits that did not effectively impact the code smell host file.

Using these metrics as a lifetime measure, we built the survival function for each code smell type. Specifically, we used the non-parametric estimator Kaplan Meier [10] to generate survival models as a function of #Days and #Effective commits, respectively.

To push forward the analysis, we report on the survival curves per code smell type. This allows us to compare the survival of the $8$ studied Android code smells and investigate their differences. Moreover, we checked the statistical significance of these differences using the Log Rank test [21]. This non-parametric test is appropriate for comparing survival distributions when the population includes censored data. We used a $95\,\%$ confidence interval—*i.e.*, $\alpha = 0.95$, with a null hypothesis assuming that the survival curves are the same.

For implementing this analysis, we respected the outline defined by Syer *et al.* [52]. We used the Python package Lifelines [14]. We used `KaplanMeierFitter` with the option `ci_show=False` to omit confidence interval in the curves. Also, we used `pairwise_logrank_test` with `use_bonferroni=True` to apply the Bonferroni [55] correction and counteract the problem of multiple comparisons.

| *Element* | *Metric* | *Description* |
|---|---|---|
| Project size | #Commits | (int): the number of commits in the project. |
| | #Developers | (int): the number of developers contributing to the project. |
| | #Classes | (int): the number of classes in the project. |
| Release | #Releases | (int): the number of releases in the project. |
| | Cycle | (int): the average number of days between the project releases. |
| Code smell | Linted | (boolean): true if the code smell is detectable by Android Lint. |
| | Priority | [1-10]: this metric is only valid for *Linted* code smells. It presents the priority given to the code smell in Android Lint. |
| | Granularity | (categories): the level of granularity of the code smell's host entity, namely: inner class, method, or class level. |

TABLE III: Metrics for *RQ2*.

*2) RQ2: What are the factors that impact the survival of Android-specific code smells?:* In this research question we investigated the impact of project size, releasing practices, and code smell properties on the survival chances of code smell instances. For this purpose, we defined the metrics presented in Table III.

*Project size :* we specifically analysed the size in terms of #Commits, #Developers, and #Classes.

*Releasing practices:* We analysed the impact of the metrics #Releases and Cycle on the survival rates. In this analysis we paid careful attention to the relevance of the studied apps for a release inspection. In particular, we manually checked the timeline of each app to verify that it used releases all the way. We excluded apps that did not use releases at all, and apps that used them only at some stage. For instance, the Chanu app [42] only started using releases in the last 100 commits. Its first $1,337$ commits do not have any release. 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 with our study artifacts [19]. It is also worth noting that as Android apps are known for continuous delivery and releasing [7], [37], we considered in this analysis both minor and major releases. This allows us to perform a fine-grained study with more releases to analyse.

*Code smell:* unlike project and release metrics, the values of code smell metrics are determined by the code smell type. Specifically, the values of Linted and Priority are defined by Android Lint [5]. Android Lint is able to detect four of our code smells, namely LIC, HMU, UIO, and IOD. It attributes a priority level from 1 to 10 that describes the importance of these code smells, where 1 is the least important and 10 is the most important. As for Granularity, it is determined by the code smell host entities that are presented in Table I. In this respect, the entities `Activity` and `View` both represent the granularity level of class. For more clarification, we report in Table IV the values of these three metrics per code smell type.

| *Code smell* | *Linted* | *Priority* | *Granularity* |
|---|---|---|---|
| LIC | True | 6 | Inner Class |
| MIM | False | – | Method |
| NLMR | False | – | Class |
| HMU | True | 4 | Method |
| UIO | True | 3 | Class |
| UHA | False | – | Class |
| IOD | True | 9 | Class |
| UCS | False | – | Method |

TABLE IV: The values of *Linted*, *Popularity*, and *Granularity* per code smell type.

To investigate the impact of the defined metrics on code smells survival, we used survival regression and stratified survival analysis.

*Survival regression:* We used this approach to analyse the impact of numerical metrics—*i.e.*, #Commits, #Developers, #Classes, #Releases, and Cycle. The survival regression allows us to regress different covariates against the lifetime variable. The most popular regression technique is Cox's proportional hazard model [13]. Cox's model has three statistical assumptions:

1) *proportional hazards*: the effects of covariates upon survival are constant over time;
2) *linear relationship*: the covariates make a linear contribution to the model;
3) *independence*: the covariates are independent variables.

We assessed these assumptions on our dataset and found that our metrics do not respect the proportional hazard assumption. That is, the impact of our numerical metrics on survival are not constant and evolved over time. Thus, we opted for the alternative technique of Aalen's additive model, which allows time-varying covariate effects [4]. The Aalen's model defines the hazard function by:

$$\lambda(t|x) = b_0(t) + b_1(t) * x_1 + ... + b_n(t) * x_n$$

Where $x_i$ refers to the studied covariates and $b_i(t)$ is a function that defines the regression coefficient over time. In our study, the covariates are the numerical metrics and the regression coefficients describe their impact on the survival of code smell instances. For the interpretation of these coefficients, we should note that the hazard function can also be defined as $\lambda(t) = 1 - S(t)$. This means that an increase in the hazard function implies a decrease in the survival one. Consequently, the metrics that have a positive hazard regression coefficient, systematically decrease the survival chances of the studied code smells and *vice versa*.

For implementation, we used `AalenAdditiveFitter` from Lifelines package [14]. This implementation estimates $\int b(t)\,\mathrm{d}t$ instead of $b(t)$. We used the function `smoothed_hazards_()` with the parameter $bandwidth = 100$ to get the actual hazard coefficients ($b(t)$).

*a) Stratified survival analysis:* We followed this approach to analyse the impact of categorical metrics, namely Linted, Priority, and Granularity. To assess the impact of these metrics, we relied on the same technique used for *RQ1*, Kaplan Meier. Specifically, we computed the survival function for each metric and category. Then, we compared the survivals among the different categories of each metric using the Log Rank test. For instance, we computed two survival curves for the metric Linted, one for Linted=True and another for Linted=False. Then, we compared the two curves using the Log Rank test.

## III. Results Analysis

For the sake of clarity, we present the numbers of analysed code smells before introducing the results of our research questions. Table V reports, for each code smell type, the numbers of instances that were introduced in our dataset (#Instances), the number of instances that were removed (#Removed), and the percentage of removal ($\%Removal = \frac{\#Removed}{\#Instances}$).

| | LIC | MIM | NLMR | HMU | UIO | UHA | IOD | UCS | All |
|---|---|---|---|---|---|---|---|---|---|
| ***#Instances*** | 98,751 | 72,228 | 4,198 | 3,944 | 514 | 267 | 93 | 18 | 180,013 |
| ***#Removed*** | 70,654 | 67,777 | 2,526 | 2,509 | 305 | 147 | 66 | 11 | 143,995 |
| ***%Removal*** | 71 | 93 | 60 | 63 | 59 | 35 | 70 | 61 | 79 |

TABLE V: Number and percentage of code smell removals.

### A. RQ1: For how long do Android-specific code smells survive in the codebase?

| | LIC | MIM | NLMR | HMU | UIO | UHA | IOD | UCS | All |
|---|---|---|---|---|---|---|---|---|---|
| ***Q1*** | 41 | 117 | 136 | 111 | 400 | 300 | 60 | 16 | 52 |
| ***Med*** | 370 | 602 | 765 | 625 | 1,007 | 1,516 | 347 | 395 | 441 |
| ***Q3*** | 1,536 | 1,978 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 1,422 | $\infty$ | 1,691 |

TABLE VI: Kaplan Meier survival in days.

*1) Survival in days:* Figures 2 and 3 show the results of Kaplan Meier analysis. To explicit our results, we also present the survival distribution in Table VI. Figure 2 illustrates the survival curves of the 8 studied code smells in terms of days. The figure indicates that the survival probabilities differ considerably depending on the code smell type. Indeed, 3,000 days after introduction, code smells like LIC and MIM have almost no chances to be alive, whereas code smells like UHA and UIO still have 40 % chances to be alive. This disparity is confirmed by Figure 3 and Table VI. We can observe that, on average, after 441 days, 50 % of all the code smells are still present in the codebase. However, this median value increases significantly among the code smells MIM, NLMR, HMU, UIO, and UHA. 50 % of these code smells are still alive after more than 600 days. We assessed the statistical significance of these differences with the Log Rank test. While we only report the significant results, the full summary of the pairwise Log Rank test between the 8 code smells is available with our artifacts [19]. Overall, the results allowed to reject the null hypothesis assuming that the survival curves of the 8 code smells are the same—*i.e.*, $p - value < 0.05$. We confirmed that the survival order shown in Table VI is significant. More specifically, we found the following:

- UIO and UHA are the code smells that survive the longest. It takes around 3 years to remove 50 % of their instances ($median > 1000\ days$);
- NLMR, HMU, and MIM also have a high survival tendency with an average of 2 years ($median > 600\ days$);
- The least surviving code smells are IOD, LIC, and UCS. 50% of their instances are removed after only one year ($median > 300\ days$).

> 50 % of the instances of Android-specific code smells stay alive in the codebase for more than 441 days.

| | LIC | MIM | NLMR | HMU | UIO | UHA | IOD | UCS | All |
|---|---|---|---|---|---|---|---|---|---|
| ***Q1*** | 3 | 1 | 3 | 1 | 2 | 3 | 2 | 3 | 3 |
| ***Med*** | 10 | 6 | 11 | 8 | 7 | 16 | 3 | 11 | 9 |
| ***Q3*** | 34 | 33 | 38 | 50 | 44 | 64 | 7 | 11 | 34 |

TABLE VII: Kaplan Meier survival in effective commits.

*2) Survival in effective commits:* Figures 4 and 5 reports on the results of Kaplan Meier analysis with effective commits. The figures show that in the 100 effective commits that follow the code smell introduction, most of the instances are removed. We also observe that there are slight differences between the survival tendencies of the 8 code smells. Indeed, Table VII and the Log Rank test results show the following:

- UHA is the longest surviving code smell. It takes 16 effective commits to remove 50 % of its instances and even after 64 effective commits 25 % of its instances are still alive ($median = 16$ and $Q3 = 64$);
- IOD is the least surviving code smell. 75 % of its instances are removed after only 7 commits on the host file ($Q3 = 7$);
- The other code smell types only have slight survival differences. In average, their instances survive for 9 effective commits and 75% of them disappear after 30 commits.

> 75% of the instances of Android-specific code smells are removed after 34 commits on the host file.

### B. RQ2: What are the factors that impact the survival of Android-specific code smells?

*1) Survival regression:* Figure 6 shows the results of Aalen's additive analysis for the project metrics #Commits, #Developers, and #Classes. It is worth noting that the three metrics are not statistically independent in our dataset. Thus, we measured their regression coefficients separately to avoid the correlation bias. From the figure, the first thing that leaps to the eye is that the three metrics have a positive impact on the hazard of code smells. Indeed, the curve values are positive during all the code smell lifetime. Thus, the project size negatively impacts the survival possibility, which means the bigger the project is, the less the code smells survive in the codebase. Overall, the positive impact on the hazard is

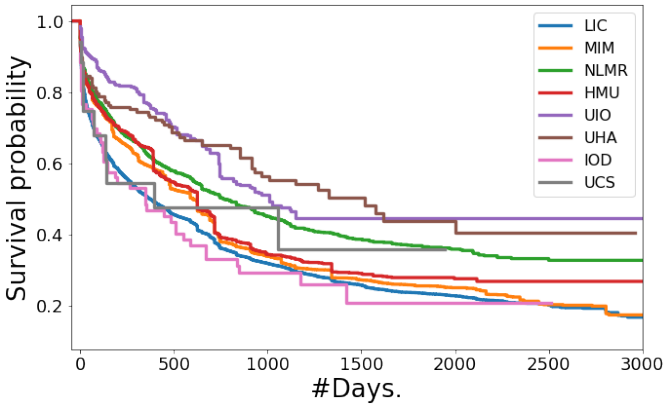


Fig. 2: Survival curves in days

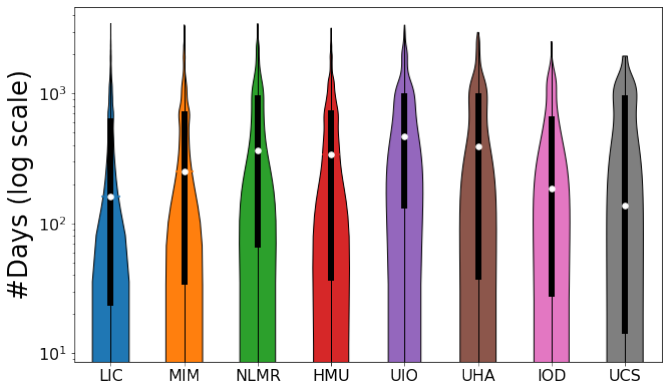


Fig. 3: Code smell lifetime in terms of days.

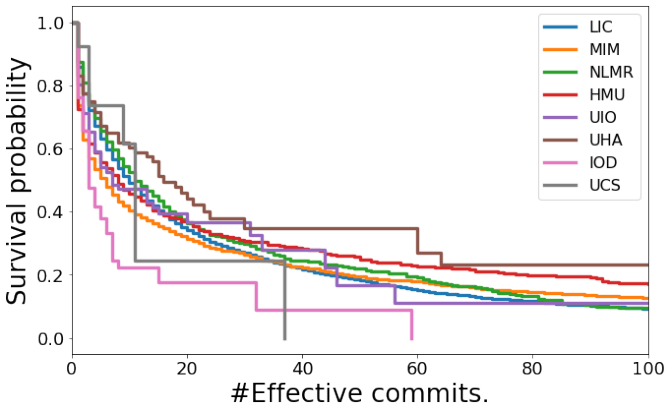


Fig. 4: Survival curves in effective commits.

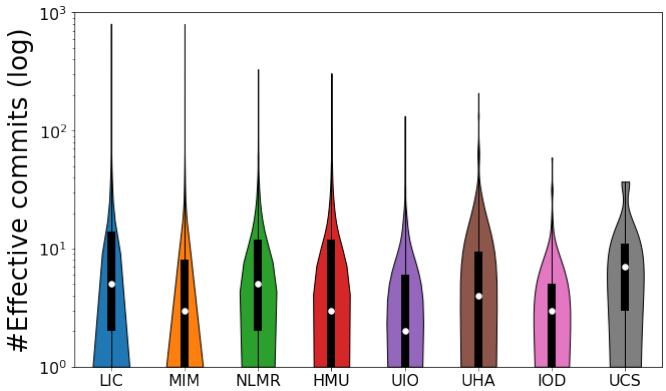


Fig. 5: Code smell lifetime in terms of effective commits.

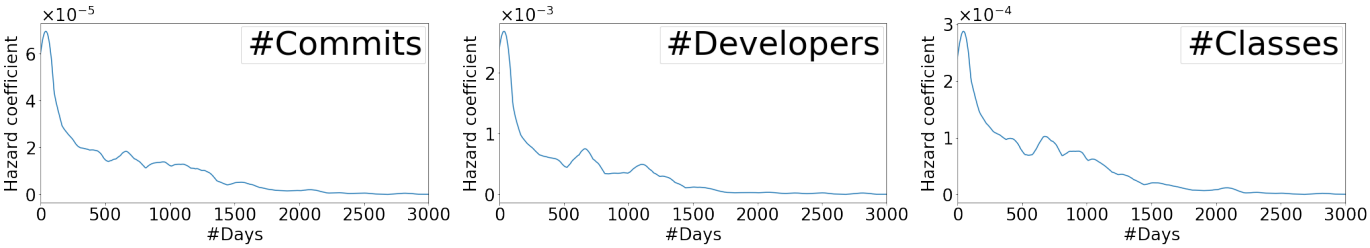


Fig. 6: The estimated hazard coefficients for #Commits, #Developers, and #Classes.

more present in the first days after the code smell introduction. Indeed, the curves of the three metrics have their highest values in the first 500 days. In the days after, the hazard rates drop significantly to stabilise after $1,500$ days. This aligns with the survival curves observed in *RQ1* where the average survival was less than 500 days.

Figure 7 shows the results of Aalen's additive analysis for the release metrics #Releases and Cycle. These two metrics are not highly correlated in our dataset $|r| < 0.3$. Hence, we can analyse their impact simultaneously with a multivariate regression.
We observe from Figure 7 that the hazard coefficients for the two variables are positive along the code smell lifetime. This shows that the two metrics have a positive impact on the hazard function. That is, increasing the number of releases and the releasing cycle tend to shorten the code smell lifetimes. Interestingly, Figure 7 also indicates that the cycle has more impact on the survival rates than the number of releases. Accordingly with the units, this means that adding one day to the average releasing cycle has more impact on survival rates than adding one release to the whole project.

> Code smells disappear faster in projects that are bigger in terms of commits, developers, and classes. Projects with longer releasing cycles and more releases also manifest shorter code smell lifetimes.

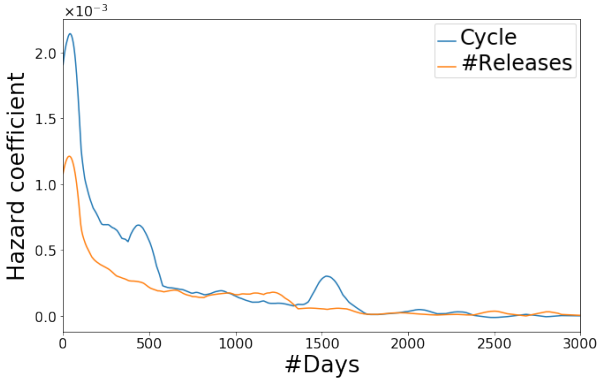


Fig. 7: The hazard coefficient of #Releases and Cycle.

*2) Stratified survival analysis:* Figure 8 compares the survival curves for code smells based on the metric *Linted*. The figure shows that the survival curve of *Linted* code smells is always under the curve of other code smells. This means that code smells that are present in Android Lint have less survival chances than other code smells. The Log Rank test confirmed the statistical significance of this observation with $p - value < 0.05$. This allows us to reject the null hypothesis assuming that the two curves are the same.

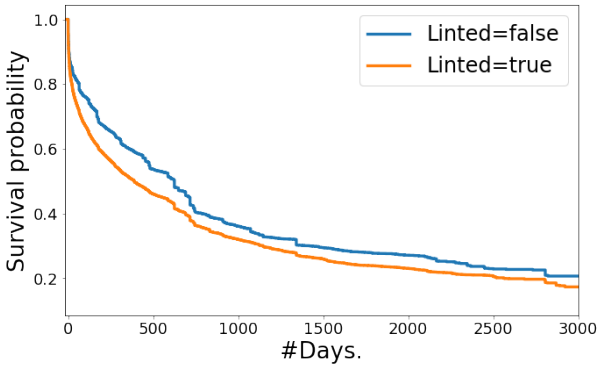


Fig. 8: The impact of the presence in the linter on survival.

> Code smells detected by Android Lint are removed from the codebase faster than other types of code smells.

Figure 9 shows the results of Kaplan Meier analysis stratified with the metric Priority. It compares the survival curves for code smells depending on their priority on Android Lint. Overall, we observe that code smells with higher priorities have less survival chances. Indeed, the code smells with the highest priority—9—have the lowest survival chances. Their survival curve is always below other curves. On the other hand, code smells with the lowest priority—3—survive longer than other code smells. The Log Rank test results showed that all these differences are statistically significant with a 95 % confidence interval. Hence, we can confirm that the more a code smell is prioritised, the less its survival chances are.

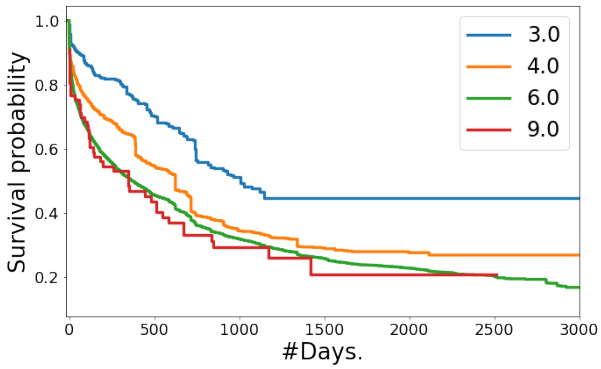


Fig. 9: The impact of linter priority on code smell survival.

> Code smells that are prioritised by Android Lint have less survival chances. They are removed faster than other code smells.

Figure 10 compares the survival curves of code smells depending on their granularity. The figure shows that code smells hosted by inner classes have the least survival chances, they disappear before other code smells. We also observe that code smells hosted by classes survive way more than code smells hosted by methods or inner classes. The Log Rank resulted in $p - values < 0.005$ for all the pairwise comparisons between the three curves. As a result, we can reject the hypotheses assuming that the survival curves of different granularities are the same.

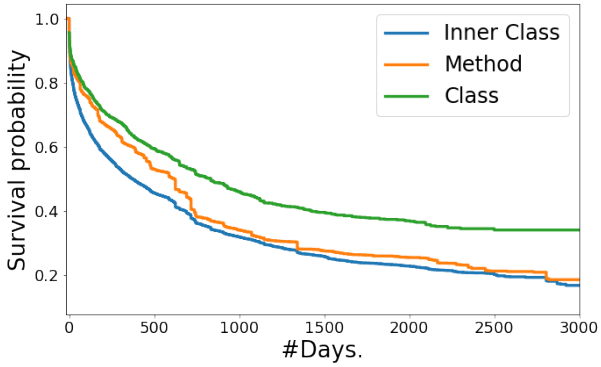


Fig. 10: The impact of granularity on code smell survival.

> The code smell granularity has an important impact on its survival chances. Code smells hosted by classes survive significantly more than smells hosted by methods and inner classes.

## IV. Discussion

### A. Discussion

Compared to the code smell longevity reported in previous studies [8], [29], [46], [54], Android code smells have a significantly shorter lifespan. Effectively, the study of Tufano *et al.* [54] reported that the median survival of OO code

smells in Android apps is over $1,000$ days and $1,000$ effective commits. In our study, the only Android code smells that reach this average longevity are UIO and UHA. All other Android code smells have a median survival of $400$ days. This shows that Android code smells are by far less persistent than OO code smells. This can be due to the nature of Android code smells, which lead them to be introduced and removed faster. Indeed, Android code smells are rather low granularity instances and they can be removed accidentally in general. For instance, a code smell like HashMap Usage is caused by the instantiation of a `HashMap` collection. Therefore, any instance of this code smell can be accidentally introduced or removed by only modifying one instruction. On the other side, OO code smells tend to be more sophisticated and may require extensive modifications to refactor them. Interestingly, our results of the granularity analysis align with this hypothesis. Indeed, the comparison demonstrated that code smells that are hosted by entities of higher granularity (class) remain longer in the source code. This confirms that, when the code smell is at a low granularity level, it becomes easier to remove and thus disappear faster from the source code.

Another surprising finding is that Android code smells have a very short lifespan in terms of effective commits. This finding highlights a huge gap between the number of days and effective commits needed to remove the code smell instances, $400$ days *vs.* $9$ effective commits. This contrast can be due to the size and complexity of Android apps. Indeed, in large software systems, the modifications are diffused across different software components. Hence, a code smell can survive in such systems for years if its host entity is not frequently modified. In this way, the survival in terms of days becomes much higher than the survival in terms of effective commits.

Our survival regression analysis showed that Android code smells disappear faster in projects that are bigger in terms of commits, developers, and classes. This observation can be due to the activity and dynamic of these projects. Indeed, bigger projects that have more contributors are potentially more active and thus are subject to more modifications. This dynamism makes these projects more prone to code smell removals. As for releases, our results showed that code smells survive less in projects that adopt more releases and longer releasing cycles. The two observations may seem contradictory as we would expect a high number of releases to imply shorter cycles. However, in practice our results showed that the two variables do not have a strong negative correlation. This means that to fasten code smell removals, developers should release frequently without going as far as to shrink releasing cycles.

Finally, our results showed that code smells detected and prioritised by Android Lint are removed faster than other code smells. This can be a direct or an indirect effect of Android Lint. Indeed, when the linter is adopted in the development process, it directly detects code smells from the source code and encourage the developer to remove them. Moreover, the linter can also make developers aware of the code smells by means of communication. In this way, even developers who disabled the linter in their *Integrated Development Environment* (IDE) may still be aware about these code smells.

To sum up, our work takes a step forward in the study of mobile code smells and opens up perspectives for new research directions. Concretely, it has the following implications:

- The high releasing frequency in the mobile ecosystem [37] does not necessarily promote bad development practices. Android code smells survive less in projects with more releases. That being said, very short releasing cycles can also be a factor that favours code smell persistence. We invite future studies to inspect in depth the impacts of releasing trends on other aspects of code smells like introductions and removals.
- The short lifespans of Android code smells incite us to hypothesise that their removal is not intended and rather accidental. We invite future works to investigate this hypothesis and inspect the actions leading to the removal of Android code smells.
- We confirm the benefits of using linters to detect performance issues in mobile apps [17]. We encourage the community to work on approaches and tools that allow the detection and refactoring of larger sets of mobile code smells.
- Our results align with previous findings about the efficiency of integrating software quality tools in the development workflow [12], [17], [26], [50]. We invite future works to invest in the integration of tools like ADOCTOR and PAPRIKA in the IDE.
- We confirm the importance of priority and severity indicators in static analysis tools [9], [12], [17], [26]. We encourage tool makers to adopt such measures in their software quality tools.

### *B. Threats To Validity*

**Internal Validity**: For this study, one main major threat could be errors-in-variables bias. In our case, this can be caused by false assessment of code smell lifetimes by associating code smell introductions or removals to the wrong commits. This can occur in situations where code smells are introduced and removed gradually or when the change history is not accurately tracked. However, as explained previously, we consider in this study only objective code smells that can be introduced or removed in a single commit. Moreover, the validation showed that, by considering branches and file renamings, SNIFFER accurately tracks code smell introductions and removals ($F1 - score = \{0.97, 0.96\}$).

**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], [32]. 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 therefore 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 validity might be affected by the gap between the concept of code smells in theory and the detection performed by PAPRIKA. However, the definitions adopted by PAPRIKA have been validated by Android developers with a precision of (0.88) and a recall of (0.93) [22], [24]. On top of that, our validation showed that PAPRIKA is also effective in the dataset under study.

**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]. We paid attention not to violate the assumptions of the performed statistical tests. We are also using non-parametric tests that do not require making assumptions about the distribution of the data. Finally, we did not make conclusions that cannot be validated with the presented results.

## V. RELATED WORK

*Code Smells in Mobile Apps:* With the advent of mobile apps as new software systems, many works have studied mobile-specific code smells. Reimann *et al.* [47] proposed a catalog of 30 quality smells dedicated to Android. These code smells cover various aspects like implementations, user interfaces or database usages. After the definition of these code smells, many research works proposed tools and approaches for detecting them [25], [27], [43], [47]. Reimann *et al.* offered a tool called REFACTORY for the detection and correction of code smells [49]. Afterwards, Hecht *et al.* [24] proposed PAPRIKA, a tooled approach that detects OO and Android code smells in Android apps. Also, Palomba *et al.* [43] proposed another tool, called ADOCTOR, able to identify 15 Android-specific code smells from the catalog of Reimann *et al.*. Other studies focused on assessing the performance impact of these code smells on app performance [11], [23], [44]. In particular, Palomba *et al.* [44] 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. To cope with bad practices in mobile apps, a few works proposed refactoring solutions [33], [40]. Notably, Morales *et al.* [40] proposed EARMO, an energy-aware refactoring approach for mobile apps. By analysing 20 open-source apps, they showed that refactoring antipatterns can decrease significantly energy consumption. Other studies aimed to understand the phenomenon of code smells in mobile apps. Mannan *et al.* [34] compared the presence of well-known OO code smells in 500 Android apps and 750 desktop applications in Java. They observed that the distribution of code smells in Android is more diversified than in desktop applications Our study complements these studies by analysing an unaddressed aspect of mobile code smells, which is lifespan and persistence in software history.

*Code Smell Survival:* To the best of our knowledge, no work has investigated mobile-specific code smells in the change history. The closest work to our study is the one of Tufano *et al.* [54], which analysed the change history of 200 open-source projects (including 70 Android apps) to understand the evolution of OO code smells. They found that OO code smells are rarely removed, only 20% of their instances were removed during the observed software history. They also observed that 50% of code smells instances persisted in the codebase for more than 1,000 days and 1,000 commits from their introduction. Several other studies investigated the longevity of OO code smells instances in the software history [8], [29], [46]. In particular, Peters and Zaidman [46] conducted a case study on 7 open-source systems to investigate the lifespan of code smells and the refactoring behavior of developers. They found that, on average, code smell instances have a lifespan of approximately 50 % of the examined revisions. Kim *et al.* [30] studied the genealogy of *code clones* in two Java open-source projects. They found that 54% to 72% of the removed instances disappeared in around 8 check-ins out of over 160 check-ins. Arcoverde *et al.* [8] conducted an explanatory survey about the longevity of code smells. They found that code smells survive for long times because developers continuously postpone refactoring operations.

## VI. CONCLUSION

We presented in this paper the first large-scale empirical study that investigates the lifespan of mobile-specific code smells in the change history. We analysed 8 Android code smells, 324 Android apps, $255k$ commits, and $180k$ code smells instances. This study resulted in several findings:

**Finding 1:** 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.

**Finding 2:** Android code smells disappear faster in bigger projects with more commits, developers, and classes.

**Finding 3:** The high releasing frequency in the mobile ecosystem does not necessarily promote long code smell lifespans.

**Finding 4:** Android code smells that are detected and prioritised by Android Lint tend to disappear before other code smell types.

These findings highlight important challenges for researchers and tool makers. In particular, we encourage the community to work on approaches and tools that allow the detection and integration of larger sets of mobile code smells. On top of the research implications, this paper provides a comprehensible replication package that includes the collected data for this paper with the used tools and scripts [19]. We highly encourage the community to build on our findings and perform further studies on 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] O. O. Aalen. A linear regression model for the analysis of life times. *Statistics in medicine*, 8(8):907–925, 1989.

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

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

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

[8] R. Arcoverde, A. Garcia, and E. Figueiredo. Understanding the longevity of code smells: preliminary results of an explanatory survey. In *Proceedings of the 4th Workshop on Refactoring Tools*, pages 33–36. ACM, 2011.

[9] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 470–481. IEEE, 2016.

[10] J. M. Bland and D. G. Altman. Survival probabilities (the kaplan-meier method). *Bmj*, 317(7172):1572–1580, 1998.

[11] 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.

[12] M. Christakis and C. Bird. What developers want and need from program analysis: an empirical study. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 332–343. IEEE, 2016.

[13] D. R. Cox. Regression models and life-tables. *Journal of the Royal Statistical Society: Series B (Methodological)*, 34(2):187–202, 1972.

[14] C. Davidson-Pilon. Lifelines. https://lifelines.readthedocs.io/en/latest/Survival%20Analysis%20intro.html, 2014. [Online; accessed January-2019].

[15] M. Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[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] 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.

[18] 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.

[19] S. Habchi, R. Rouvoy, and N. Moha. Study artifacts. https://figshare.com/s/9977eeabd9265d713bdb, 2019. [Online; accessed January-2019].

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

[21] D. P. Harrington and T. R. Fleming. A class of rank test procedures for censored survival data. *Biometrika*, 69(3):553–566, 1982.

[22] 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.

[23] 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.

[24] 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.

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

[26] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013.

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

[28] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 75–84. IEEE, 2009.

[29] 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.

[30] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 187–196. ACM, 2005.

[31] D. G. Kleinbaum and M. Klein. *Survival analysis*, volume 3. Springer, 2010.

[32] 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.

[33] Y. Lin and D. Dig. Refactorings for android asynchronous programming. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 836–841. IEEE, 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] S. McIlroy, N. Ali, and A. E. Hassan. Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. *Empirical Software Engineering*, 21(3):1346–1370, 2016.

[38] R. Minelli and M. Lanza. Software analytics for mobile applications–insights and lessons learned. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 144–153. IEEE, 2013.

[39] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A. Le Meur. Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20–36, 2010.

[40] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol. Earmo: An energy-aware refactoring approach for mobile apps. *IEEE Transactions on Software Engineering*, 2017.

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

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

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

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

[50] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 598–608. IEEE Press, 2015.

[51] G. SCM. Git tagging. https://git-scm.com/book/en/v2/Git-Basics-Tagging, 2014. [Online; accessed January-2019].
[52] M. D. Syer, M. Nagappan, B. Adams, and A. E. Hassan. Replicating and re-evaluating the theory of relative defect-proneness. *IEEE Transactions on Software Engineering*, 41(2):176–197, 2015.
[53] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 403–414. IEEE Press, 2015.
[54] 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.
[55] E. W. Weisstein. Bonferroni correction. 2004.