An Empirical Study of the Performance Impacts of Android Code Smells

Geoffrey Hecht, Naouel Moha, Romain Rouvoy

An Empirical Study of the Performance Impacts of Android Code Smells

Geoffrey Hecht
University of Lille / Inria, France
Université du Québec à Montréal, Canada
geoffrey.hecht@inria.fr

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

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

ABSTRACT

Android code smells are bad implementation practices within Android applications (or apps) that may lead to poor software quality, in particular in terms of performance. Yet, performance is a main software quality concern in the development of mobile apps. Correcting Android code smells is thus an important activity to increase the performance of mobile apps and to provide the best experience to mobile end-users while considering the limited constraints of mobile devices (*e.g.*, CPU, memory, battery). However, no empirical study has assessed the positive performance impacts of correcting mobile code smells.

In this paper, we therefore conduct an empirical study focusing on the individual and combined performance impacts of three Android performance code smells (namely, *Internal Getter/Setter*, *Member Ignoring Method*, and *HashMap Usage*) on two open source Android apps. To perform this study, we use the PAPRIKA toolkit to detect these three code smells in the analyzed apps, and we derive four versions of the apps by correcting each detected smell independently, and all of them. Then, we evaluate the performance of each version on a common user scenario test. In particular, we evaluate the UI and memory performance using the following metrics: *frame time*, *number of delayed frames*, *memory usage*, and *number of garbage collection calls*. Our results show that correcting these Android code smells effectively improve the UI and memory performance. In particular, we observe an improvement up to 12.4% on UI metrics when correcting *Member Ignoring Method* and up to 3.6% on memory-related metrics when correcting the three Android code smells. We believe that developers can benefit from these results to guide their refactoring, and thus improve the quality of their mobile apps.

Keywords

Android, code smells, metrics, mobile computing, performance.

ACM ISBN .
DOI:

1. INTRODUCTION

Along the last years, the development of mobile applications (or apps) has reached a great success. In 2013, Google Play Store[1] reached over 50 billion app downloads [4] and is estimated to reach 200 billion by 2017 [7]. As of January 2016, date of the writing of this paper, Google Play store counts almost 2 millions of Android apps in the market of which 11% are classified as low quality apps [5]. This success is partly due to the adoption of established *Object-Oriented* (OO) programming languages, such as Java, Objective-C or C#, to develop these mobile apps. However, mobile apps differ significantly from traditional software systems [26] since, for their development, it is necessary to consider the specificities of mobile platforms. For example, apps heavily rely on external libraries and reuse of classes [26, 30, 38]. Dedicated approaches and studies addressing software quality of mobile apps should therefore be developed while considering the mobile specificities and the constraints on resources like memory, CPU, screen sizes, etc.

Moreover, the ever-increasing user requirements and popularity of mobile apps have led mobile developers to implement, maintain, and evolve apps rapidly and under pressure. Hence, mobile developers may adopt bad design and implementation practices, also known as *code smells* [18]. The presence of code smells may lead to poor software quality, thus hindering the evolution of apps and degrading the quality of the software [34] and its end-user experience.

In particular, the presence of code smells can be imposed by the underlying framework [24, 36], but also lead to resource leaks (CPU, memory, battery, etc.) [14], thus preventing the deployment of sustainable solutions. These performance problems can induce an important impact on user experience, and around 18% of Android apps suffered or are still suffering of these problems [25]. Performance is then a main software quality concern in the development of mobile apps. Thus, correcting code smells can contribute to improve performance and user experience without impacting the app behavior. This is the reason why the correction of code smells, such as *Internal Getter/Setter* (IGS) or *Member Ignoring Method* (MIM), is mentioned in the Android documentation in the performance tips section [3].

However, so far, only the local impact on performance—*i.e.*, the CPU time of a method or the memory usage of one variable—has been assessed [3, 10]. Therefore, there is no empirical evidence that correcting these code smells can significantly improve the software quality and end-user ex-

[1] `https://play.google.com/store`

perience. Moreover, the correction of these code smells is time consuming and can introduce some minor drawbacks with regard to the app evolution. For example, the correction of HashMap Usage might become ineffective when `HashMaps` contain more than hundreds of items [10]. Also to the best of our knowledge, these code smells were only evaluated on devices using the Dalvik runtime but not using the recent ART runtime, which already concerns one third of the active Android devices [1]. Since ART is able to perform more optimization than Dalvik [8] and does not use the same garbage collection strategies [2], the positive impact of correcting the code smells on ART might be less important. Developers may therefore be reluctant to correct these code smells for unpredictable benefits.

Therefore, this paper is intended to support mobile developers in choosing to correct or not the code smells we studied. Our goal is to determine if there is a positive impact on performance metrics related to user experience when code smells are corrected. For this purpose, we correct the studied code smells in two open source Android apps and then evaluate the impact of the correction on four performance metrics.

The rest of the paper is organized as follows. We provide some background information describing the studied code smells and relevant metrics in Section 2. Section 3 presents the design of our study and Section 4 discusses the obtained results. Section 5 presents related works on the impact of code smells. Section 6 concludes our study and outlines some avenues for future works.

2. BACKGROUND

In this section, we briefly introduce the three code smells under study in this paper and we outline their expected impact in terms of performance. Then, we present and qualify the usage of the metrics collected to measure the performance.

2.1 Studied Code Smells

We choose the following code smells because they are reported in the Android framework documentation to have a *local* positive impact on performance and because it is possible to correct them without affecting the behavior of the app. Moreover, a previous analysis with the Paprika tool has shown that these code smells are commonly present in Android apps [21], and thus could be corrected in substantial proportion. In theory, they are straightforward to correct and do not introduce any side effect on performance. Therefore, a good practice would be to avoid them at all costs. However, their correction can increase the maintenance effort or may reduce performance in future releases of the app for some very specific changes. Even if these drawbacks are highly hypothetic, they must be taken into account before correction by developers.

Internal Getter/Setter (IGS) is an Android code smell that occurs when a field is accessed, within the declaring class, through a getter (`var = getField()`) and/or a setter (`setField(var)`). This indirect access to the field may decrease the performance of the app. The usage of IGS is a common practice in OO languages like C++, C# or Java because compilers or virtual machines can usually inline the access. However, there is only simple inlining for Android [9] and, consequently, the usage of a trivial getter or setter is often converted into a virtual method call, which makes the operation at least three times slower than a direct access. This code smell can be corrected by accessing the field directly within a class (`var = this.myField`, `this.myField = var`) and declaring the getter and setter methods in the public interface. Correcting IGS with refactoring is therefore a way to increase the performance of the method accessing a field [3, 14]. Of course, non-trivial getters/setters, as illustrated in Listing 1, are not concerned by this code smell. Therefore, a possible drawback of fixing an IGS can happen when a trivial getter/setter is modified into a non-trivial getter/setter in a future version of an app.

Listing 1: Example of non-trivial getter in SoundWaves Podcast app

```
public String getURL() {
    String itemURL = "";
    if (this.url != null && this.url.length() > 1)
        itemURL = this.url;
    else if (this.resource != null &&
         this.resource.length() > 1)
        itemURL = this.resource;
    return itemURL;
}
```

Member Ignoring Method (MIM). In Android, when a method does not access an object attribute or is not a constructor, it is recommended to use a static method in order to increase performance. The static method invocations are about 15%–20% faster than dynamic invocations [3]. It is also considered as a good practice for readability since it ensures that calling the method will not alter the object state [3, 14]. However, there is one possible side effect in terms of inheritance since all the extending classes have to declare or to refer to the same static methods. Listing 2 is an example of MIM.

Listing 2: Example of Member Ignoring Method in SoundWaves Podcast app

```
private boolean
     animationStartedLessThanOneSecondAgo(long
     lastDisplayed) {
  return System.currentTimeMillis() - lastDisplayed <
       1000 && lastDisplayed != -1;
}
```

HashMap Usage (HMU). The Android framework provides `ArrayMap` and `SimpleArrayMap` as replacements from standard Java `HashMap`. They are supposed to be more memory-efficient and trigger less garbage collection with no significant difference on operations performance for maps containing up to hundreds of values [10]. So, unless a complex map for a large set of objects is required, the use of `ArrayMaps` should be preferred over the usage of `HashMap` for Android apps. Therefore, creating small `HashMap` instances can be considered as a code smell [10, 20]. However, a performance degradation of using a `HashMap` can occur when facing an unpredicted growth of the map. An example of HashMap Usage is provided in Listing 3.

Listing 3: Example of HashMap Usage in SoundWaves Podcast app

```
if (itemMap == null) {
    itemMap = new HashMap<>();
    for (int i = 0; i < ItemColumns.ALL_COLUMNS.length;
         i++) {
      itemMap.put(ItemColumns.ALL_COLUMNS[i],
          ItemColumns.ALL_COLUMNS[i]);
    }
}
```

2.2 Selected Metrics

The impact of the previous code smells has already been assessed by Google engineers at a local level using micro-benchmarks [3, 10]. However, there is no proof that this local positive effect is still significant when the whole app is considered. In particular, the impact of the correction of these code smells on the user experience is never considered. For this reason, we choose to use performance metrics that are known to be related to user experience.

Frame time: The frame time is the time taken by the operating system to draw one frame of the app. To provide the best user experience, it is recommended to reach and keep 60 *frames per second* (FPS) on Android. In the case of a FPS drop, the user may feel that the app is not smooth and unresponsive during animations, such as scrolling a window. Unresponsiveness of apps is often reported in user reviews [25], which proves that it affects user experience. To reach 60 FPS, all input, computing, network, and rendering actions should be executed in less than 16 ms *per* $frame$ [12, 19]. Therefore, the frame time is a global performance metric bound to user experience that can be affected by all kinds of optimizations including the correction of code smells.

Number of delayed frames: On Android, a frame is delayed or dropped when it takes more than 16 ms to be drawn. The frame buffer is only posted to the screen every 16 ms, which means that a delayed frame will not be displayed before 32 ms even if it takes only 17 ms. Such an app, which runs constantly at 17 ms, will therefore be capped at 30 FPS instead of the recommended 60 FPS [12, 19]. This metric is directly derived from the frame time and provides some additional information about the user experience.

Memory usage: The memory usage of an app measured in KB can affect the entire system. Indeed, memory is limited on mobile devices (512 MB is a common configuration) and the system is running multiple apps at the same time. When the system runs out of memory and no memory is freed after triggering garbage collection, the system has to free the memory used by currently running apps. This leads to the obligation to completely reload apps in the memory at the next utilization. Thus, the loading time of apps is increased and therefore, in order to allow the user to quickly switch between apps, each app should consume a minimum of memory [11, 19]. Excessive memory usages are also reported by users in their reviews when they affect their experience [25].

Number of garbage collection calls: Garbage collection is an automatic memory management, which allows the system to free the memory of objects that are no longer used by apps. Under the Dalvik virtual machine, the garbage collection can take up to 20 ms on fast devices [19]. Numerous calls to the garbage collector due to a bad usage of memory in an app can then lead to a degradation of the global performance of the app. In particular, it can lead to an increase in the number of delayed frames and thus affect the user experience [11, 19].

Relation with code smells: HMU is directly related to the last two metrics by its definition. IGS and MIM have an effect on the execution time of a method. They may have an effect on the first two metrics, since the *frame time* does not only concern GPU time, but also CPU time. The *frame time* includes the time spent in each method executed in the main UI thread, but not in the background.

3. STUDY DESIGN

This section reports the design of our study, which aims to bring out the positive performance impact of correcting IGS, MIM, and HMU in two open source Android apps. In this purpose, we address the four following research questions:

RQ$_1$: *Does the correction of IGS, MIM or HMU improve the UI drawing performance?*

RQ$_2$: *Does the correction of IGS, MIM or HMU improve the memory performance?*

RQ$_3$: *Does the correction of the three code smells improve more significantly the UI drawing performance compared to the correction of only one code smell?*

RQ$_4$: *Does the correction of the three code smells improve more significantly the memory performance compared to the correction of only one code smell?*

RQ$_5$: *Does the correction of the code smells still have an impact with ART runtime instead of Dalvik ?*

3.1 Objects

The first open source mobile app used in this study, called *SoundWaves Podcast*[2], is a podcast client, which allows users to search, download, and listen to podcasts on their Android devices. It relies on iTunes and gPodder for the search. This study is based on the version 0.112 available on GitHub[3], which was the latest version available at the time this study was conducted. This app counts around 520 classes (including internal classes) and 2,672 methods.

The second open source app of this study is *Terminal Emulator for Android*[4], an app that allows users to access the Android's built-in Linux command line shell. We used the last version 1.0.70 also available on GitHub[5]. *Terminal Emulator* counts 141 classes and 978 methods.

These apps were selected after the analysis of a set of 50 random open-source apps available from F-Droid[6]. We choose these apps because all the three examined code smells

[2] `https://play.google.com/store/apps/details?id=org.bottiger.podcast`
[3] Soundwaves Github: `https://github.com/bottiger/SoundWaves`
[4] `https://play.google.com/store/apps/details?id=jackpal.androidterm`
[5] Terminal Emulator Github: `https://github.com/jackpal/Android-Terminal-Emulator`
[6] F-Droid: `https://f-droid.org/`

Table 1: Experimental Versions

Version	Corrected Code Smells
V_0	None
V_1	Internal Getter/Setter (IGS)
V_2	Member Ignoring Method (MIM)
V_3	HashMap Usage (HMU)
V_4	All (IGS + MIM + HMU)

were present in a quantity allowing us to correct them manually since the code smell correction is a time consuming task. Indeed, we detected 60 code smells: 24 IGS, 29 MIM, and 7 HMU in *SoundWaves*. We detected 20 code smells in *Terminal Emulator*: 6 IGS, 10 MIM, and 4 HMU in *SoundWaves*. This study focuses only on the main package of these two apps: `org.bottiger.podcast` and `jackpal.androidterm`, respectively.

All included third-party libraries, such as Picasso[7], are excluded from our study since they are imported via Gradle, and thus the developers do not have access to their code.

We run all experiments on a Motorola Moto G XT1032 8GB with Android version 4.4.4 (KitKat). It has a Qualcomm Snapdragon 400 processor Quad core 1.2 GHz processor, 1 GB RAM and a 4.5 inches display with resolution of 720 x 1280 pixels. It can be considered as a mid-range smartphone for year 2015.

3.2 Design

To assess the impact of the IGS, MIM and HMU, we detected and corrected these code smells in the *SoundWaves Podcast* and *Terminal Emulator for Android* apps and we tested them using the scenarios described in the next Section 3.3. The metrics used in this study are collected during 60 executions of these scenarios. In total, we obtained five versions of each app, as described in Table 1. V_0 is the version of the app downloaded from GitHub with no modifications. V_1, V_2, and V_3 are derived from V_0 by correcting IGS, MIM and HMU, respectively. In version V_4, all code smells are corrected. We performed our experiments only on two apps since producing and instrumenting each version manually is a time consuming task. The 60 scenarios executions for one version can take more than five hours. *SoundWaves Podcast* was instrumented with both ART and Dalvik runtimes to answer RQ_5.

3.3 Procedure

Detection and correction of the code smells: First, we detected the three smells in the apps by performing a static analysis with the Paprika tool [21]. We obtained a list of methods and classes concerned by the three code smells. Then, we corrected manually each Android smell to obtain the versions presented in Table 1. It should be noted that the HMU is corrected using `ArrayMap` of the package `android.support.v4.util` to ensure that V_3 and V_4 keep the compatibility with the same versions of Android compared to V_0. The implementation in `android.util` is only available for API level 19 and superior. The list of detected and corrected code smells is available online.[8]

As presented in Table 2, we detected and corrected a total of 80 code smells in 49 classes for both apps. So these classes can contain more than one code smell and some methods can include more than one IGS or HMU. In *SoundWaves*, `HashMaps` are declared twice in methods and 5 times in classes while 3 of 4 maps in *Terminal emulator* are declared in classes, but they are always used in at least one method. Obviously, they can be only one MIM per method, according to the definition given in Section 2. As expected by the number of classes in the apps, *Terminal Emulator* contains less code smells than *Soundwaves*.

We instrumented the original versions, V_0 of each app, to count how many times each code smell is invoked as presented in Table 2. These instrumented versions are slightly different from V_0 because of the integrated code for instrumentation and therefore, the average number of drawn frames might be quite different between all versions. So the numbers of invocations may vary, and the given values are just there to give an order of magnitude.

We can observe that IGS and MIM are invoked frequently in *Soundwaves*, respectively 3145 and 4361, while this is only the case for IGS in *Terminal Emulator*, *i.e.* 2831 and 28. This high frequency is due to the fact that some of the concerned methods are directly or indirectly called from `onDraw()` methods of views and activities. They can be invoked up to 60 times per second. HMU is less frequent, however the `HashMap` may stay in the memory for a certain amount of time (depending on the scenario and garbage collections). Consequently, its effect on memory usage and on the number of garbage collections also remain over time.

User scenario test: After correcting the detected code smells, each version was executed 60 times using a Python script that launches the Robotium test automation framework.[9] This framework allows developers to write black-box UI tests for native and hybrid Android apps. For *SoundWaves*, we defined a user scenario test composed of 185 steps (including around 90 wait operations) that navigates through most of the functionalities, views and menus of the app. In particular, this scenario includes the search of a podcast using keywords and the subscription to a podcast. The scenario duration is about 235 seconds. The *Terminal Emulator* scenario contains 201 steps (with 91 wait operations) and lasts for about 167 seconds. This scenario includes the usage of commands, such as *ls*, as well as the usage of multiple windows. The scenarios are run via the `android.test.InstrumentationTestRunner` included in the Android framework, which ensures a minimum overhead during tests. The functionality to download and play a podcast is not included in the test of *SoundWaves* since it depends on the network quality and uses the default Android player to play the music; this scenario is outside the scope of the app. The wait operations are used to ensure that all the views and the included images are fully loaded before using any functionality, so that the scenario is carefully reproduced for each execution. We also used simulated gestures, such as scrolling or swiping, instead of direct accesses to views or activities, to be as close as possible from a real user experience. These scenarios are available online[10].

Collection of metrics: As depicted in Figure 1, a Python script orchestrates the whole experiment by launching ADB

[7] `http://square.github.io/picasso`

[8] List of corrected code smells: `http://sofa.uqam.ca/paprika/mobilesoft16.php#CodeSmells`

[9] `http://www.robotium.org`

[10] Robotium scenarios: `http://sofa.uqam.ca/paprika/mobilesoft16.php#Scenarios`

Table 2: Number of code smells corrected, number of entities concerned and average number of code smells invocations during the scenarios

		# corrected	in # methods	in # classes	average # invocations
SoundWaves	IGS	24	21	11	3145
	MIM	29	29	21	4361
	HMU	7	4	4	21
	Total	60	53	33	7527
Terminal	IGS	6	2	4	2832
	MIM	10	10	8	29
	HMU	4	2	2	12
	Total	20	8	16	2873

(Android debugger) commands and the Robotium test to collect the metrics in the form of logs stored into different files. The `logcat` command is running continuously during each experiment. It allows us to collect the number of garbage calls for the complete system only, so other running processes may affect this metric. The explicit calls to the garbage collector are excluded since there is no such call in the instrumented apps. The `dumpsys gfxinfo` command gives us performance information related to the last 120 displayed frames for a given app. It allows us to extract the number of delayed frames and the frame time. This command is executed every second to ensure that no frame is missed since the maximum FPS allowed by Android is 60 FPS. Finally, the memory usage is obtained via the command `dumpsys memoryinfo`, which allows us to investigate the memory usage of an app. We only consider the private memory usage of the process.

Minimization of extraneous factors: To ensure that every test runs in similar conditions, we perform an explicit call to the garbage collector using `System.gc()` during the setup phase of each test. Moreover, the app process is killed before each iteration of the test. There is no SIM card in the phone and functionalities, such as Bluetooth, data network, GPS, and notifications of other apps are deactivated. We only use the WiFi network to allow the app to perform online search of podcasts. The instrumented app is the only running app during each experiment, and the legacy Android processes and services are still running in the background. The app is installed on the device memory since there is no SD card in the phone, and therefore we only use internal memory, which is faster than SD memory.

3.4 Variables and Hypotheses

Independent Variables: The numbers of IGS, MIM, and HMU corrected in each version of the app and the device runtime environment (Dalvik or ART) are the independent variables of our study.

Dependent Variables: The dependent variables correspond to the metrics related to the performance of the app in terms of UI drawing and memory usage. These metrics are presented in Section 2 and are used to investigate our research questions. They are collected during the execution of the scenarios presented in Section 3.3.

Hypotheses: To answer our four research questions, we formulate the following null hypotheses, which we applied to the two apps, where V_0, V_x ($x \in \{1 \dots 3\}$) and V_4 are the different versions of the app, as described in Table 1:

- $\text{HR}^{FT}_{V0\,Vx}$: There is no difference between the *frame time* (FT) of versions V_0 and V_x;
- $\text{HR}^{DF}_{V0\,Vx}$: There is no difference between the *number of delayed frames* (DF) of versions V_0 and V_x;
- $\text{HR}^{MU}_{V0\,Vx}$: There is no difference between the *memory usage* (MU) of versions V_0 and V_x;
- $\text{HR}^{GC}_{V0\,Vx}$: There is no difference between the *number of garbage collection calls* (GC) of versions V_0 and V_x;
- $\text{HR}^{FT}_{V4\,Vx}$: There is no difference between the *frame time* (FT) of versions V_4 and V_x;
- $\text{HR}^{DF}_{V4\,Vx}$: There is no difference between the *number of delayed frames* (DF) of versions V_4 and V_x;
- $\text{HR}^{MU}_{V4\,Vx}$: There is no difference between the *memory usage* (MU) of versions V_4 and V_x;
- $\text{HR}^{GC}_{V4\,Vx}$: There is no difference between the *number of garbage collection calls* (GC) of versions V_4 and V_x.

3.5 Analysis Method

We performed the Mann-Whitney U test [32] to test hypotheses $\text{HR}^{FT}_{V0\,Vx}$, $\text{HR}^{MU}_{V0\,Vx}$, $\text{HR}^{FT}_{V4\,Vx}$, and $\text{HR}^{MU}_{V4\,Vx}$ since we have hundreds of values for the frame time and memory usage metrics. We also computed the Cliff's δ effect size [29] to quantify the importance of the difference between metric values for hypotheses $\text{HR}^{DF}_{V0\,Vx}$, $\text{HR}^{GC}_{V0\,Vx}$, $\text{HR}^{DF}_{V4\,Vx}$, and $\text{HR}^{GC}_{V4\,Vx}$ since the metrics are ordinal values. We selected the Cliff's δ effect size because it is reported to be more robust and reliable than the Cohen's d effect size [16]. All the tests are performed using a 95% confidence level—*i.e.*, p-value < 0.05. Mann-Whitney U test is a non-parametric statistical test that assesses whether two independent distributions are the same or if one distribution tends to have higher values. Non-parametric statistical tests make no assumptions about the distributions of the metrics. Cliff's δ is a non-parametric effect size measure, which represents the degree of overlap between two sample distributions [29]. It ranges from -1 (if all selected values in the first group are larger than the second group) to $+1$ (if all selected values in the first group are smaller than the second group). It equals 0 when two sample distributions are identical [15].

Interpreting the Effect Sizes: Cohen's d is mapped to Cliff's δ via the percentage of non-overlap, as shown in Table 3 [29]. Cohen [17] states that a medium effect size represents a difference likely to be visible to a careful observer, while a large effect is significantly larger than medium. We also use the average values of all experiments to make comparison between versions.

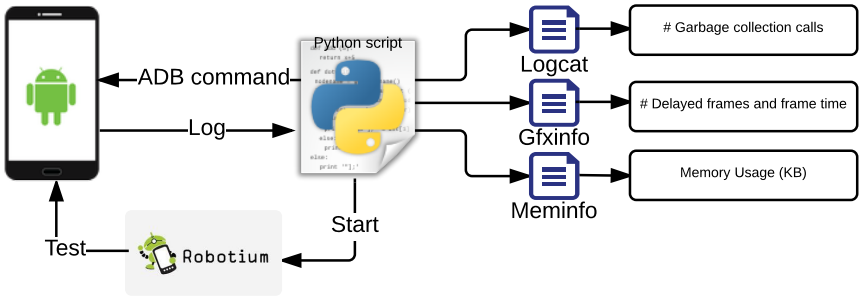

Figure 1: Study process managed by a Python script

Table 3: Mapping Cohen's d to Cliff's δ.

Cohen's Standard	Cohen's d	% of Non-overlap	Cliff's δ
small	0.20	14.7%	0.147
medium	0.50	33.0%	0.330
large	0.80	47.4%	0.474

4. CASE STUDY RESULTS

This section reports and discusses the results we obtained to answer our research questions.

4.1 Overview of the results

Figure 2 reports the average memory usage over time for all versions of *SoundWaves*. The time period reported in Figure 2 corresponds to the last 120 seconds of the scenarios executions.We can already observe that the memory usage over time for all versions follows similar curves. All the versions seem to have relatively the same performance for this metric with a slight benefit for V_3 and V_4. The results are similar for *Terminal Emulator*.

The average values of the 60 experiments for each version are presented in Table 4. First of all, we can observe that, most of the time, the correction of code smells improves—*i.e.*, decreases—the average values of metrics, but surprisingly in some case the values increase. For example, the correction of IGS (V_1) slightly increases the memory usage in both apps. In both apps, V_4 is the best version regarding memory usage. Overall, V_4 performs well in all metrics and tends to cumulate the impact effects of the correction of all code smells. For other versions, the results are different between the two apps. V_2 is the best performing version for *SoundWaves* concerning UI metrics. However, for *Terminal Emulator*, V_2 is better only for the frame time metric, while V_1 outperforms V_2 for the delayed frame metric. V_4 is the best version for GC calls in *Terminal Emulator*, while for *SoundWaves*, it is for V_3. We expected such differences since the number and location of corrected code smells and the used scenarios are different for each app. Nevertheless, this means that for some other apps and other scenarios, the impact of the correction may vary. Thus, in the rest of this paper, according to the metrics, we will mostly focus on the app where we can observe a significant impact to show the potential impact of correcting code smells. Although we can already observe a positive impact in favor of all derived versions, it is insufficient to decide on the significance of these results. Therefore, we compute the percentage difference in Tables 5 and 7 and the Mann-Whitney U test and Cliff's δ effect size in Tables 6 and 8.

Table 4: Metrics average values on 60 experiments.

App	Version	Frame time	Memory usage (KB)	Delayed frame	GC calls
SoundWaves	V_0	5.36	39,071.78	54.08	485.68
	V_1	5.32	39,119.85	50.27	494.07
	V_2	**5.30**	39,152.48	**47.40**	473.70
	V_3	5.33	38,920.65	51.73	**467.90**
	V_4	5.31	**38,887.32**	48.60	468.13
Terminal	V_0	3.83	13,223.11	39.78	78.22
	V_1	3.81	13,227.36	**39.18**	76.83
	V_2	**3.80**	13,217.89	40.07	77.43
	V_3	3.82	13,227.78	40.67	77.12
	V_4	3.82	**13,141.58**	40.27	**76.25**

We can already observe in Tables 6 and 8 that all differences between versions are not always significant, in particular for frame time and memory usage. For delayed frames and GC calls in Tables 5 and 7, the differences are more important than for other metrics and often in favor of the derived versions for both apps. However, we can observe that for GC calls in *SoundWaves*, the Cliff's δ effect size (see Table 6) gives a large negative impact whereas the average values (see Table 5) tend to show a positive impact for all derived versions. Indeed, taken individually, the number of GC calls for V_0 are often and in a large proportion slightly smaller than the results obtained for the other versions, which explains the results of Cliff's δ effect size. However, V_0 has also extreme values where the difference with other versions is very high, which give a positive impact for derived versions on average.

We also collected the results for *SoundWaves* using ART runtime instead of Dalvik. GC calls were not collected since they are not any more relevant for our comparison due the improvement in ART [6]. Only two partial calls to the garbage collector were logged during our scenario execution using ART runtime. These results are not presented in de-

Table 5: Metrics percentage differences of average values between versions for SoundWaves.

	Frame time	Memory usage	Delayed frame	GC calls
V_0,V_1	-0.61%	0.12%	-7.06%	1.73%
V_0,V_2	**-1.12%**	0.21%	**-12.36%**	-2.47%
V_0,V_3	-0.50%	-0.39%	-4.35%	**-3.66%**
V_0,V_4	-0.83%	**-0.47%**	-10.14%	-3.61%
V_1,V_4	-0.22%	-0.59%	-3.32%	-5.25%
V_2,V_4	0.30%	-0.68%	2.53%	-1.18%
V_3,V_4	-0.33%	-0.09%	-6.06%	0.05%

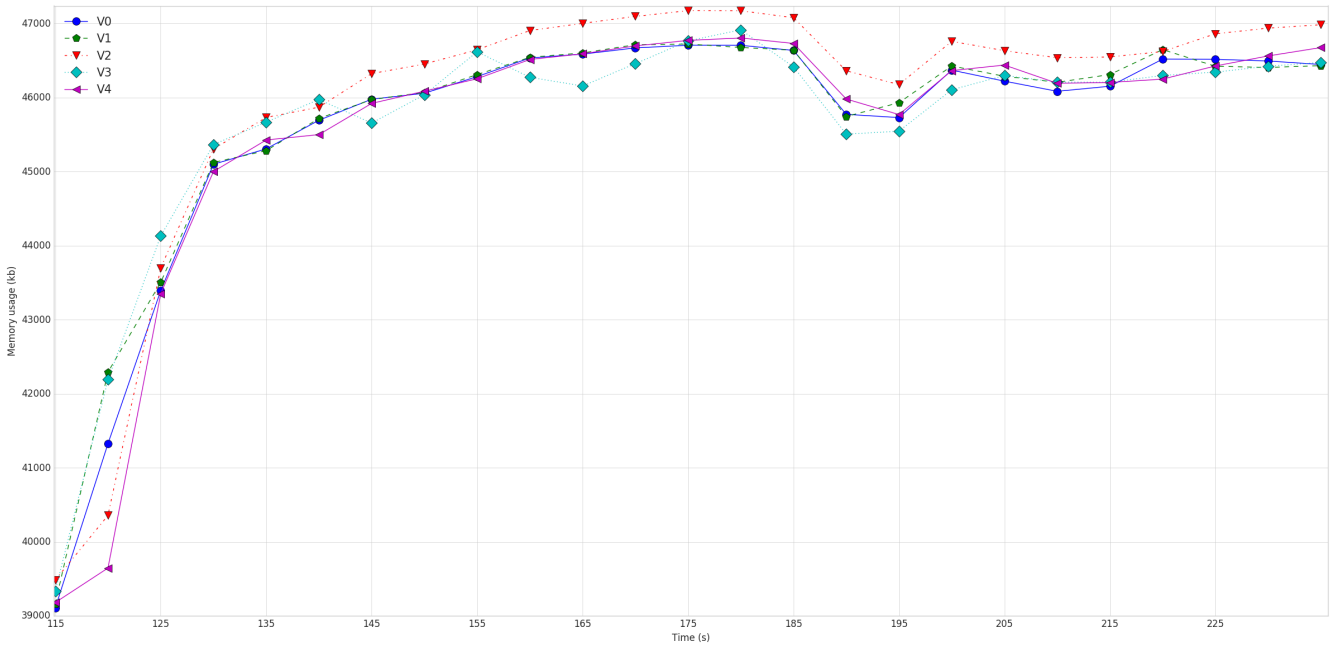

Figure 2: Average memory usage over time for all versions of SoundWaves (last 120 seconds).

Table 6: Mann-Whitney U test and Cliff's δ effect size (S for small, M for medium, L for Large) for all versions of SoundWaves.

	Mann-Whitney U test		Cliff's delta	
	Frame time	Memory usage	Delayed frame	GC calls
V_0,V_1	0.065	0.786	**-0.314 (S)**	**0.708 (L)**
V_0,V_2	0.744	**<0.05**	**-0.285 (S)**	**0.729 (L)**
V_0,V_3	0.633	0.303	**-0.190 (S)**	**0.729 (L)**
V_0,V_4	0.618	0.123	**-0.364 (M)**	**0.728 (L)**
V_1,V_4	0.183	0.312	0.0002	**0.539 (L)**
V_2,V_4	0.475	**<0.05**	-0.019	**0.520 (L)**
V_3,V_4	0.953	0.250	**-0.149 (M)**	0.140

Table 7: Metrics percentage differences of average values between versions for Terminal Emulator.

	Frame time	Memory usage	Delayed frame	GC calls
V_0,V_1	-0.54%	0.03%	**-1.51%**	-1.77%
V_0,V_2	**-0.58%**	-0.04%	0.71%	-1.00%
V_0,V_3	-0.24%	0.04%	2.22%	-1.41%
V_0,V_4	-0.20%	**-0.62%**	1.21%	**-2.51%**
V_1,V_4	0.34%	-0.65%	2.76%	-0.76%
V_2,V_4	0.39%	-0.58%	0.50%	-1.53%
V_3,V_4	0.04%	-0.65%	-0.98%	-1.68%

tails in this paper but are available online [11]. In summary, the impact of delayed frames and memory usage is not significant for all versions. However, there is a significant but slight improvement of around 1% in all versions for the frame time. Compared to Dalvik there is an average increase in memory usage of around 20% but an improvement of 8% for frame time and 18% less delayed frames.

The remaining results are discussed in more details in the following sections while answering the research questions.

[11]Results of the study: `http://sofa.uqam.ca/paprika/mobilesoft16.php#Results`

Table 8: Mann-Whitney U test and Cliff's δ effect size (S for small) for all versions of Terminal Emulator.

	Mann-Whitney U test		Cliff's delta	
	Frame time	Memory usage	Delayed frame	GC calls
V_0,V1	**<0.05**	0.934	-0.031	**-0.282 (S)**
V_0,V2	**<0.05**	0.797	0.060	-0.059
V_0,V3	0.122	0.985	0.102	**-0.151 (S)**
V_0,V_4	**<0.05**	**<0.05**	0.070	**-0.147 (S)**
V_1,V_4	0.487	**<0.05**	0.113	0.075
V_2,V_4	0.841	**<0.05**	0.045	-0.121
V_3,V_4	0.406	**<0.05**	-0.018	-0.05

4.2 RQ1: Does the correction of IGS, MIM or HMU improve the UI drawing performance?

Internal Getter/Setter: Results provided in the previous tables tend to show a non-significant or very slight impact on frame time for both apps so we accept $\text{HR}^{FT}_{V0\,V1}$. Concerning the delayed frame metric, we reject $\text{HR}^{DF}_{V0\,V1}$ and confirm that the correction of IGS does reduce the number of delayed frames, as we can observe for *SoundWaves* in Table 5 and Table 6. Moreover, although the improvement is not significant for *Terminal Emulator* (see Tables 7 and 8), we can observe that V_1 is even though the best performing version for this metric with -1.51%. The observed results show that most of the drawn frames are not impacted by the correction explaining the results on frame time, but the frames that are concerned tend to be delayed less often. Moreover, for these frames, some getters/setters are called directly or indirectly within the method `onDraw()` of some views. Therefore, they participate directly in the drawing of frames and can be called up to 60 times per seconds.

Member Ignoring Method: For the same reasons, we support $\text{HR}^{FT}_{V0\,V2}$, but reject $\text{HR}^{DF}_{V0\,V2}$ and we assess the pos-

itive impact of the correction of MIM on the number of delayed frames. The results for *Terminal Emulator* are also non significant for delayed frames, but this can be explained by the few invocations of concerned methods. Here again, in *SoundWaves* the call from `onDraw()` of views might be responsible for this effect.

HashMap Usage: We also accept $HR^{FT}_{V0\,V3}$, but we reject $HR^{DF}_{V0\,V3}$ considering the significant results in *SoundWaves*. This can be explained by the side effect of the reduced numbers of garbage collections, which are known to increase the number of delayed frames as explained in Section 2.

Our results show that the correction of IGS, MIM, and HMU does not impact the frame time, but significantly reduces the number of delayed frames, and thus favorably contributes to improve the UI drawing performance.

4.3 RQ2: Does the correction of IGS, MIM or HMU improve the memory performance?

Internal Getter/Setter: The results on both apps show no impact of correcting IGS on memory usage, hence we accept $HR^{MU}_{V0\,V1}$. However, we have conflicting results between apps concerning garbage collection calls. In *SoundWaves*, there is a significant increase of 1.73% whereas there is a significant decrease of 1.77% for *Terminal Emulator*. Hence, we reject the null hypothesis $HR^{GC}_{V0\,V1}$. However, we are not able to determine if the effect will be always positive or negative. This impact could appear surprising since the definition of IGS never mentions any effect on memory. However, it can be explained by the fact that the Dalvik Virtual Machine puts in cache virtual *call sites* to perform optimizations [9]. Since the correction of the IGS removes some virtual calls, it may affect the garbage collector behavior. Further investigations are necessary to understand in which cases this side effect is either positive or negative.

Member Ignoring Method: Here again, the results for *Terminal Emulator* are non significant due to the few invocations of concerned methods. For *SoundWaves*, the difference in memory usage is significant for the Mann-Whitney U test, but it is only of 0.21% (see Table 5), hence we accept $HR^{MU}_{V0\,V2}$. We reject $HR^{GC}_{V0\,V2}$ but we cannot determine if the effect is positive or negative since, as explained in the overview (see Section 4.1), Cliff's delta and the average values are conflicting. We are not aware of the Dalvik Virtual Machine specificities that can explain the effect on garbage collection. However, our hypothesis is that it could be linked to the usage of implicit object parameters, which refer the instance of a class when non-static methods are called.

HashMap Usage: Here again, the difference in memory usage is very slight for both apps (-0.39% and 0.05%) and not significant and we accept $HR^{MU}_{V0\,V3}$. Most of the HMU smells corrected concern very small maps of less than 50 items and this explains that the difference is not visible on this metric. We also have conflicting results concerning garbage collection calls in *SoundWaves*, but V_3 has the best average of all versions with a decrease of 3.66%. Moreover, there is a significant positive effect in *Terminal Emulator*. Hence, we can reject $HR^{GC}_{V0\,V3}$ for HMU. Even if there is only a few of these code smells that were corrected, the impact is visible on garbage collection.

Overall, our results show that the correction of HMU reduces the number of garbage collection calls, and thus contributes to improve the memory performance. However, this is not the case for IGS and MIM.

4.4 RQ3: Does the correction of the three code smells improves more significantly the UI drawing performance compared to the correction of only one code smell?

Concerning the frame time, we support all null hypotheses $HR^{FT}_{V4\,Vx}$ since the correction of the three code smells does not improve significantly the frame time performance. This is due to the fact that there is not real impact of correcting the code smell on this metric, as shown in the answer of RQ1.

Based on the results of Table 5 and Table 6, we reject $HR^{DF}_{V4\,V0}$ and $HR^{DF}_{V4\,V3}$, but we accept $HR^{DF}_{V4\,V1}$ and $HR^{DF}_{V4\,V2}$. In both apps, the effect of code smell correction tends to cumulate and thus even if it is not the best version, it is performing better than most versions.

In our study, the correction of the three code smells does not outperform the best version with only one code smell corrected but it outperforms all the other versions.

4.5 RQ4: Does the correction of the three code smells improve more significantly the memory performance compared to the correction of only one code smell?

For the memory usage, the very good results of V_4 in memory usage for *Terminal Emulator* allow us to reject all null hypotheses $HR^{MU}_{V4\,Vx}$. V_4 is also the best version with -0.62% for this metric for *SoundWaves* even if the improvement is only slight compared to other versions as we can observe in Figure 2. This result is interesting since we did accept all other hypotheses on memory usage in RQ2. It is probably due to the very slight effect of all corrections on memory usage that are significant when cumulated, but not significant when taken separately.

For the number of garbage collections, we reject all null hypotheses $HR^{GC}_{V4\,Vx}$ except $HR^{GC}_{V4\,V3}$. The performance on this metric for V_3 is on par or slightly better than V_4. We make the same observations as for Section 4.1 for *SoundWaves*: taken individually, most of the number of garbage collection calls for V_0,V_1,V_2 and V_3 are slightly smaller, but on average the other values are significantly larger. Therefore, we cannot confirm or reject a positive effect for this app, however the effect is always positive for *Terminal Emulator*.

As for memory, the correction of the three code smells does improve the memory performance compared to all other versions.

4.6 RQ5: Does the correction of the code smells still have an impact with ART runtime instead of Dalvik ?

Our results on *SoundWaves* available online[11] tend to show that there is no impact of correcting the code smells on delayed frames and memory usage. Indeed, the results are not significant or there is no visible effect size on these metrics. Therefore, we accept all hypotheses $HR^{DF}_{V0\,Vx}$ and $HR^{MU}_{V0\,Vx}$ for ART. Concerning frame time, the p-value of the Mann-Whitney U test is always inferior to 0.05 meaning that the impact is significant. Hence, we reject all hypotheses $HR^{FT}_{V0\,Vx}$. However, this impact is very slight since there is only an improvement of around 1% for all code smells. This difference has then no significant effect on the number of delayed frames.

The correction of the studied code smells only has a slight impact on frame time but no impact on the other metrics with ART.

4.7 Threats to Validity

In this section, we discuss the threats to validity of our study based on the guidelines provided by Wohlin *et al.* [37].

Construct validity threats concern the relation between theory and observations. In this study, they could be due to measurement errors. That is the reason why we did several experiments and used averages instead of instant values. Moreover, we tried to reduce as much as possible external factors as explained in our procedure in Section 3.3. We also performed our experiments on a real device instead of an emulator since GPU and CPU emulations are still experimental [13].

Internal validity threats concern the causal relationship between the treatment and the outcome. During our study, we were very careful about the interpretation of our results and the relationship with the study process. In particular, when only one code smell was corrected we tried to explain what could be the cause of the observed results by investigating the source code and the virtual machine process. We are aware that the correction of multiple code smells could lead to unexpected interplay, and we have been also very careful about our interpretations.

External validity threats concern the possibility to generalize our findings. Further validations should be done on different apps and with different code smells to broaden our understanding of the impact of code smells on the performance of Android apps. In the same way, the values we found are specific to the used apps, scenarios and the selected device, and thus cannot be generalized at this time. Thus, we are not assuming that our results can be used to estimate the impact of correcting a code smell. However, we believe that this paper contributes to prove that there is a global performance impact for the studied code smells.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study and our analysis. Furthermore, the scenarios, python script and the dataset used in this study are available online to leverage its reproduction[11].

Finally, the *conclusion validity* threats refer to whether the conclusions reached in a study are correct. We paid attention not to violate the assumptions of the performed statistical tests. We mainly used non-parametric tests that do not require making assumptions about the distribution of the metrics. We were also careful with our conclusion when the results on our two apps were conflicting.

5. RELATED WORK

In this section, we discuss the relevant literature focusing on the impact of correcting code smells. In particular, the closest works to our contribution studied energy consumption as a performance metric.

Li and Halfond investigated the impact of energy-saving programming practices on Android [23]. In particular, they measured the impact of implementing some performance tips on energy consumption by comparing a method with and without the tips. IGS and MIM were considered in this study and improved the energy consumption by about 33% and 15%, respectively. Moreover, they discovered that higher memory usage slightly increases the energy consumption.

Tonini *et al.* examined the performance time and energy consumption of IGS and different `for` loop syntaxes [35]. They compared the CPU time and the energy consumption by executing 30 times a portion of code with different practices including IGS and loops. They confirmed that the correction of IGS improves the execution time of calls up to 30% and reduces the energy consumption down to 27%.

Mundody and K did a similar work with CPU time and energy consumption using also IGS and different `for` loop syntaxes [27]. In a similar way, they compared different methods with and without applying the good practices. They also evaluated the effect of the correction of IGS in two Android apps, and found that the difference were significant for CPU time and energy according to the Student's t-Test [33]. However, there is no detail on the number of corrections and the process used to instrument the apps.

Sağlam studied the correlation between the presence of code smells in an app and the rating of an app [31]. In particular, he observed that apps containing the MIM code smells tend to have worst user ratings in a significant proportion. He supposed that removing these code smells may improve user ratings in the long-term. This is an indication that the presence of code smells may affect the user experience.

Although these works are relevant and represent significant contributions, they mainly focus on the energy consumption impact of Android code smells at a local level whereas the last reported work only consider the correlation between the presence of code smells and user ratings. This paper aims to supplement these previous works by studying the impact of correcting code smells on performance metrics related to user experience at the app level.

As for the definition of the code smells studied in this paper, MIM and IGS were defined by Reimann *et al.* [28]. They proposed a catalog of 30 quality smells dedicated to Android. These code smells are mainly originated from the good and bad practices documented in the Android online documentations or by developers reporting their experience on blogs. These quality smells concern various aspects like implementation, user interfaces, or database usages. They are reported to have a negative impact on properties, such as efficiency, user experience, or security.

In this paper, we use our previously developed tool, called Paprika, to detect these code smells [21, 22]. We defined the HMU as a code smell by reading the Android official documentation [10] and Android developers recommendations [20]. This code smell, as well as IGS and MIM, were integrated within Paprika as queries that can be executed on any app.

6. CONCLUSION AND FUTURE WORK

Android code smells are bad practices that may decrease the performance of an Android app, and thus affecting the user experience. While their local impact—*i.e.*, on the CPU time of a method or the memory usage of one variable—can be evaluated by micro-benchmarks, there is no proof that such micro-optimizations may have a significant impact on high-level metrics, such as the frame rendering time on a specific mobile device. In this paper, we performed series of experiments with different versions of two open source Android apps to determine if the correction of the Internal Getter/Setter (IGS), Member Ignoring Method (MIM), and HashMap Usage (HMU) code smells has a significant impact on UI and memory performance. We used the frame time and the number of delayed frames as UI metrics because they are known to be related to the user experience. Similarly, we used the memory usage and the number of garbage collection calls to determine the impact on the memory.

Our results show that the correction of these code smells can improve the previous metrics in a significant way for Dalvik runtime. In particular, the correction of MIM performs very well concerning the UI metrics in one of the studied app with 12.4% less delayed frames, whereas the correction of HMU has the most significant impact on memory performance for both apps with a reduction of 3.6% in terms of garbage collection calls. We also observed that the correction of the three code smells is a good choice to perform well on all metrics, even it can be outperformed on some metrics by the correction of only one code smell. This is due to the fact that the positive and negative impact of code smells correction cumulates. The correction of these code smells improves slightly the frame time on the recent ART runtime but has no significant effect on others metrics.

We believe that developers can benefit from our initial results to improve the performance of their apps by correcting the aforementioned code smells. In our opinion, the hypothetical drawbacks of the correction of these code smells can be disregarded compared to the potential benefits of their correction.

In the future, we plan to extend our study to investigate a broader variety of code smells and mobile apps. We also plan to automatize the time-consuming task of the code smell correction and the study process. Therefore, it will be possible to support large-scale studies on the performance impact of mobile code smells.

Acknowledgements

These researches are co-funded by Université of Lille, Université du Québec à Montréal, Inria, The *Natural Sciences and Engineering Research Council of Canada* (NSERC), *Fonds de recherche du Québec - Nature et technologies* (FQNRT) and *Programme Frontenac*. The authors thank Guillaume Connan from Polytech Nantes for his help with the experiments.

7. REFERENCES

[1] Android dashboards. `http://developer.android.com/about/dashboards/index.html`. [Online; accessed January-2016].

[2] Android memory tuning for android 5.0 and 5.1. `https://01.org/android-ia/user-guides/android-memory-tuning-android-5.0-and-5.1`. [Online; accessed January-2016].

[3] Android performance tips. `http://developer.android.com/training/articles/perf-tips.html`. [Online; accessed January-2016].

[4] Android will account for 58% of smartphone app downloads in 2013, with ios commanding a market share of 75% in tablet apps. `https://www.abiresearch.com/press/android-will-account-for-58-of-smartphone-app-down`. [Online; accessed January-2016].

[5] Appbrain stats : Number of android applications. `http://www.appbrain.com/stats/number-of-android-apps`. [Online; accessed January-2016].

[6] Art and dalvik. `https://source.android.com/devices/tech/dalvik/`. [Online; accessed January-2016].

[7] Mobile applications futures 2013-2017. `http://www.portioresearch.com/en/mobile-industry-reports/mobile-industry-research-reports/mobile-applications-futures-2013-2017.aspx`. [Online; accessed January-2016].

[8] Optimizing compiler – the evolution of art. `http://www.androidauthority.com/art-optimizing-compiler-605011/`. [Online; accessed January-2016].

[9] What optimizations can i expect from dalvik and the android toolchain? `http://stackoverflow.com/a/4930538`, 2011. [Online; accessed January-2016].

[10] Arraymap. `http://developer.android.com/reference/android/support/v4/util/ArrayMap.html`, 2015. [Online; accessed January-2016].

[11] Investigating your ram usage. `http://developer.android.com/tools/debugging/debugging-memory.html`, 2015. [Online; accessed January-2016].

[12] Testing display performance. `http://developer.android.com/training/testing/performance.html`, 2015. [Online; accessed January-2016].

[13] Using the emulator. `http://developer.android.com/tools/devices/emulator.html`, 2015. [Online; accessed January-2016].

[14] M. Brylski. Android smells catalogue. `http://www.modelrefactoring.org/smell_catalog`, 2013. [Online; accessed January-2016].

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

[16] J. Cohen. *Statistical power analysis for the behavioral sciences (rev.* Lawrence Erlbaum Associates, Inc, 1977.

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

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

[19] C. Haase. Developing for android, i: Understanding the mobile context. `https://goo.gl/KUN6XC`, 2015. [Online; accessed January-2016].

[20] C. Haase. Developing for android, ii the rules: Memory. `https://medium.com/google-developers/`

developing-for-android-ii-bb9a51f8c8b9, 2015. [Online; accessed January-2016].

[21] 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.

[22] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien. *Detecting Antipatterns in Android Apps*. PhD thesis, INRIA Lille, 2015.

[23] D. Li and W. G. Halfond. An investigation into energy-saving programming practices for android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, pages 46–53. ACM, 2014.

[24] M. Linares-Vásquez, S. Klock, C. McMillan, A. Sabané, D. Poshyvanyk, and Y.-G. Guéhéneuc. Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. In *Proc. of the 22nd International Conference on Program Comprehension*, pages 232–243. ACM, 2014.

[25] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1013–1024. ACM, 2014.

[26] 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.

[27] S. Mundody and S. K. Evaluating the impact of android best practices on energy consumption. *IJCA Proceedings on International Conference on Information and Communication Technologies*, ICICT(8):1–4, October 2014. Full text available.

[28] 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.

[29] 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.

[30] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan. Understanding reuse in the android market. In *20th International Conference on Program Comprehension (ICPC)*, pages 113–122. IEEE, 2012.

[31] Ġ. A. Sağlam. *Measuring And Assesment Of Well Known Bad Pratices In Android Application Developments*. PhD thesis, Middle East Technical University, 2014.

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

[33] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[34] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*, volume 11. Elsevier Science, 2014.

[35] A. R. Tonini, L. M. Fischer, J. C. B. de Mattos, and L. B. de Brisolara. Analysis and evaluation of the android best practices impact on the efficiency of mobile applications. In *2013 III Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 157–158. IEEE, 2013.

[36] D. Verloop. *Code Smells in the Mobile Applications Domain*. PhD thesis, TU Delft, Delft University of Technology, 2013.

[37] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer, 2012.

[38] L. Xu. *Techniques and Tools for Analyzing and Understanding Android Applications*. PhD thesis, University of California Davis, 2013.