Code Smells in iOS Apps: How do they compare to Android?

Sarra Habchi, Geoffrey Hecht, Romain Rouvoy, Naouel Moha

Code Smells in iOS Apps: How do they compare to Android?

Sarra Habchi[1], Geoffrey Hecht[1,2], Romain Rouvoy[1,3], Naouel Moha[2]
[1] Inria / University of Lille, France
[2] Université du Québec à Montréal, Canada
[3] IUF, France
sarra.habchi@inria.fr, geoffrey.hecht@inria.fr, romain.rouvoy@inria.fr, moha.naouel@uqam.ca

***Abstract*—With billions of app downloads, the Apple App Store and Google Play Store succeeded to conquer mobile devices. However, this success also challenges app developers to publish high-quality apps to keep attracting and satisfying end-users. In particular, taming the ever-growing complexity of mobile apps to cope with maintenance and evolution tasks under such a pressure may lead to bad development choices. While these bad choices, *a.k.a.* code smells, are widely studied in object-oriented software, their study in the context of mobile apps, and in particular iOS apps, remains in its infancy.**

Therefore, in this paper, we consider the presence of object-oriented and iOS-specific code smells by analyzing 279 open-source iOS apps. As part of this empirical study, we extended the PAPRIKA toolkit, which was previously designed to analyze Android apps, in order to support the analysis of iOS apps developed in Objective-C or Swift. We report on the results of this analysis as well as a comparison between iOS and Android apps. We comment our findings related to the quality of apps in these two ecosystems. Interestingly, we observed that iOS apps tend to contain the same proportions of code smells regardless of the development language, but they seem to be less prone to code smells compared to Android apps.

***Keywords*—Mobile apps; iOS; Android; software quality; code smells.**

I. INTRODUCTION

Mobile apps have to continuously evolve in order to meet the user expectations and stay ahead of the app stores competition. However, at design and code levels, this evolution usually increase the complexity and smell-proneness of mobile apps. Code smells are well-known in the *Object-Oriented* (OO) development community as poor or bad practices that impact negatively the software maintainability, even causing long-term problems [24]. The presence, the evolution, and even the impact of code smells in OO systems have been widely addressed in the literature [33], [46], [51]. Recently, Mannan *et al.* analyzed the publications related to code smells that have been published between 2008 and 2015 in top conferences in Software Engineering: ICSE, FSE, OOPSLA/SPLASH, ASE, ICSM/ICSME, MSR, and ESEM [37]. From a total of 52 papers, they found only 5 papers addressing code smells in Android and no paper related to iOS or Windows Phone. The rest 47 papers concern only code smells in desktop applications. This shows that code smells in the iOS ecosystem remains an open issue that needs to be addressed by the community.

In this paper, we study code smells in the iOS ecosystem. In particular, we first propose a catalog of 6 iOS-specific code smells that we identified from the developers' feedbacks and the platform official documentation. These smells have not been addressed in the literature before. In our proposed catalog, we also emphasize the similarities between these 6 smells and 3 Android-specific smells.

Secondly, we build on PAPRIKA [28], a tooled approach that detects OO and Android smells in Android apps. Specifically, our study exploits an extension of PAPRIKA to analyze iOS apps developed with Objective-C or Swift languages. To the best of our knowledge, PAPRIKA is the only tooled approach able to detect iOS specific code smells.

The first part of our study aims to assess the presence of OO and iOS code smells in open-source iOS apps in order to compare the proportions of code smells between apps developed in Objective-C and Swift. We analyze 103 Objective-C and 176 Swift apps to detect the code smells from our catalog.

In the second part of this study, we compare the results obtained on iOS apps with the results of the analysis of 1,551 open-source Android apps analyzed with PAPRIKA. This study aims to highlight the differences in term of smells presence between the two mobile platforms.

The objective of this study is to answer the following two research questions:

RQ1: Are OO and iOS smells present in Swift and Objective-C with the same proportion?

FINDING: Yes, we discovered that despite the differences between the two languages, code smells tend to appear with the same proportion or only a slight difference in Objective-C and Swift.

RQ2: Are code smells present in iOS and Android with the same proportion?

FINDING: No, for all code smells at the exception of the Swiss Army Knife we observed that Android apps tend to have a significantly bigger proportion code smells.

This paper is organized as follows. Section II discusses existing works on code smells detection in mobile apps. Section III introduces concepts related to the analysis of iOS apps code source and binaries. Section IV reports on the catalog of 6 iOS-specific smells. Section V describes how

we modified the PAPRIKA tooled approach to adapt it to iOS apps. Section VI reports the results of an empirical study that examines the presence of OO and iOS-specific code smells on a dataset of 103 Objective-C and 176 Swift apps. Section VII compares the results of the detection of the 6 code smells between iOS and Android apps. Finally, Section VIII summarizes our work and outlines further works to perform on this research topic.

II. RELATED WORK

In this section, we discuss the relevant literature about code smells in mobile apps and their detection. With regard to iOS apps, there are only few tools that support the analysis of code quality. However, to the best of our knowledge, none of them supports specific iOS smells. OCLINT [50] is a static analyzer that inspects C, C++ and Objective-C code, searching for code smells and possible bugs. Unlike LINT [2], which detects Android-specific code smells, OCLINT does not support iOS-specific smells. INFER [31] works also on Android and C programs, and uses the static analysis for detecting potential bugs in iOS apps. Moreover, it detects possible memory leaks in iOS, *null pointer* exceptions and resource leaks in Android. CLANG [15], SONARQUBE [62] and FAUX PAS [23] are also static analyzers that can be used to detect potential bugs in Objective-C projects, but they do not support specific iOS flaws. For Swift, the main code analyzers are TAILOR [69], SWIFT LINT [56] and LINTER SWIFTC [9]. TAILOR is a static analyzer that checks styling consistency and helps avoiding bugs. SWIFT LINT aims to enforce Swift style and conventions, and LINTER SWIFTC is a Linter plugin for syntax analyzis.

Looking to other mobile platforms, Linares-Vásquez *et al.* [35] used DECOR [46] to perform the detection of 18 different OO code smells in mobile apps built using *Java Mobile Edition* (J2ME) [53]. This large-scale study was performed on 1,343 apps and shows that the presence of code smells negatively impacts the software quality metrics, in particular metrics related to fault-proneness. They also found that some code smells are more common in certain categories of Java mobile apps. Regarding Android, Verloop [75] used popular Java refactoring tools, such as PMD [64] and JDEODORANT [72] to detect code smells, like *large class* and *long method* in open-source Android apps. They found that code smells tend to appear at different frequencies in core classes (*i.e.,* classes that inherit from the Android framework) compared to non-core classes. For example, *long method* was detected twice as many in core classes in terms of ratio. However, Android-specific code smells were not considered in these two studies.

Regarding the analysis of Android apps, there are a few tools and approaches devoted to the study of Android code smell specificities [2], [45], [60].

Reimann *et al.* [57] propose a catalog of 30 quality smells dedicated to Android. These code smells are mainly originated from the good and bad practices documented online in Android documentations or by developers reporting their experience on blogs. They cover various aspects like implementations, user interfaces or database usages. They are reported to have a negative impact on properties, such as efficiency, user experience or security. Reimann *et al.* are also offering the detection and correction of code smells via the REFACTORY tool [58]. This tool can detect code smells from an *Eclipse Modeling Framework* (EMF) model. The source code can be converted to EMF if necessary. However, we have not been yet able to execute this tool on an Android app. Moreover, there is no evidence that all the code smells of the catalog are detectable using this approach.

Android SDK integrates Lint [2], a rule-based static code analysis tool able to scan Android projects searching for potential OO and Android specific code smells (*Leaking Inner Class*, *Internal Getter/Setter*, and variants of *Init OnDraw*, or *OverDraw* among others). Lint offers the possibility of adding new rules to implement the detection of user defined code smells. However, the addition of rules requires the user to develop the detection algorithms in Java.

Gjoshevski and Schweighofer [25] used 140 Lint rules in a study that aims to analyze whether the size (in terms of lines of code) is related to the technical debt and which are the most common issues on Android apps. They analyzed 30 open source Android apps and conclude that the size, in terms of lines of code, does not impact on the technical debt and that the most common code smells in the apps under study were *Visibility modifier*, *Avoid commented-out lines of code*, and *Magic number*.

Mannan *et al.* [37] compared the presence of well-known OO code smells in 500 Android apps and 750 desktop applications in Java. They did not observe major differences between these two types of applications in terms of density of code smells. However, they observed that the distribution of code smells on Android is more diversified than for desktop applications, where the most common code smells are by far internal and external duplications.

To the best of our knowledge, the only functional approach devoted to the detection of Android code smells is PAPRIKA [29]. It models Android apps in a large-scale graph, which is explored by specific queries to detect code smells. We describe it in more details in Section V as we extended this approach to analyze iOS apps.

III. BACKGROUND ON IOS APPS ANALYSIS

This section reports on the ongoing state of practice in terms of both static and dynamic analysis in the iOS ecosystem.

A. Dynamic Analysis

Most of the existing dynamic analysis techniques work on applications and systems running on x86 architectures. While this architecture is widely used for desktop operating systems, mobile software systems rather use the ARM architecture, which is completely different [68]. Consequently, the existing dynamic analysis techniques cannot be used for analyzing iOS apps. Another constraint for the dynamic analysis of iOS applications is the general adoption of event-based graphic

user interfaces, which makes application features depending on the events triggered by the user. Moreover, a finite number of automatic runs of the app may be unable to cover all the execution paths [68]. Therefore, a relevant dynamic analysis of iOS apps requires necessarily an interaction with the graphic interface.

B. Static Analysis of Binaries

Although Objective-C is a strict superset of the C language, its binaries differ from the C/C++ ones by the use messages to support the interactions among objects. Moreover, all messages are routed by the routine `objc_msgSend`. This means every method call from an object to another induces an additional call to the method `objc_msgSend` [22]. The routing and the additional calls impact significantly the semantics of control flow graphs, and thus the results of the static analysis. Besides, resolving the real method calls in this case requires manipulations at the level of the CPU registers and type information tracking, which is highly complex [22].

C. Static analysis of Source Code

iOS applications are made available in the App Store as IPA files. With the IPA extension, the code source files are encrypted with FairPlay, a technique for numerical rights management adopted by Apple, and compressed with the ZIP format [52]. As a result, the access to the source code through these files requires decryption and reverse-engineering, which is prohibited. This implies that source code analysis can only be applied on open-source apps.

Synthesis: Regarding the current state of practices, constraints, and tools available in the iOS ecosystem to analyse mobile apps, we consider that the most relevant solution to study code smells in iOS applications consists in analysing the source code of applications published as open-source software.

IV. Code Smells in iOS Apps

This section introduces our first contribution: a catalog of iOS-specific code smells. We first explain the process we followed to mine common iOS smells, then we describe the identified smells as a catalog, and finally we highlight the similarities between these code smells and the Android ones.

A. Code Smells Identification Process

As code smells in iOS have not been addressed by the state of the art, we decided to rely on the platform official documentation and the community's knowledge to identify these smells. The sources we considered relevant for our task are:

- The Apple developer guide [7];
- Well-known web references for iOS development: Ray-Wenderlich [55] and Objc.io [49];

Using a grounded theory method [67], we manually parsed the posts of these sources and identified the problems raised by developers, as well as the critically recommended practices. Thereafter, we selected the most relevant bad practices and detailed them from data of:

- Developers blogs;
- Q&A forums like Stack Overflow [65].

Lastly, we formalized the problems using the mini-antipattern template proposed by Brown *et al*. [11].

B. Catalog of iOS Code Smells

Name: *Singleton Abuse*
Category: `Conceptual`
Problem: Singleton is one of the design patterns recommended by Apple for the development of iOS apps [5]. Developers interact often with this pattern through platform classes like `UIApplication` and `NSFileManager`. Moreover, the XCode IDE has a default code to easily generate a Singleton instance. However, this tends to encourage abuse of this pattern in inappropriate situations.
Example: A recurrent example of abusive use of Singleton is when the singleton class is used for storing global variables of the application—*e.g.* user data that are accessed by all the app classes. Having a global data in the application makes it stateful, and thus difficult to understand and debug [66].
References: [1], [19], [30], [34], [66]

Name: *Massive View Controller* (MAVC)
Category: `Conceptual`
Problem: Most of the iOS apps are designed using MVC, a design/architectural pattern that splits the application into three layers: *model*, *view*, and *controller*, and where the default role of the controller is to link the two other layers [42]. However, in iOS, app controllers tend to carry much more responsibilities than connecting the model to the views [42]. In particular, the controller handles UI events, like clicks and swipes, because it is part of the response chain. It also receives the system warnings regarding the memory state and it has to perform the necessary operations for managing the memory occupied by the app. All these additional responsibilities make the controllers massive, complex, and difficult to maintain [32].
References: [8], [13], [41], [42], [54], [63]

Name: *Heavy Enter-Background Tasks* (HEBT)
Category: `Performance`
Problem: Since version 4.0 of iOS, apps are allowed to execute tasks in the background. This possibility requires from apps some specific adjustments, like freeing the memory space or stopping some tasks, to manage the transition from front to background. These adjustments must be applied to the method `applicationDidEnterBackground:` from the `AppDelegate` class, which is called when the app moves to the background [6]. However, a problem occurs when the operations of this method last for a long time, thus exhausting the allocated execution time, and causing the completion handler to be called in order to suspend the app.
References: [6]

Name: *Ignoring Low-Memory Warning* (ILMW)
Category: `Performance`
Problem: In iOS, when the system requires more memory to perform its tasks, it sends low-memory warnings to the apps holding an important memory space via the method `did-ReceiveMemoryWarning:` of the `UIViewController` class. Every view controller should implement this method to free the unused memory space—*e.g.*, views elements that are not currently visible. However, when the method is not implemented, the view controller cannot react to the system warnings, and if the application is holding an important memory space, it will be killed by the system [3].
References: [3], [38]

Name: *Blocking The Main Thread* (BTMT)
Category: `Performance`
Problem: In iOS, the UIKit is directly tied to the main thread, so all the graphical user interface interactions are in this thread and are impacted by the execution time of the other operations sharing it [4]. Therefore, every heavy processing in the main thread makes the UI completely unresponsive. The operations that often block the main thread are identified by [39] as:

- The synchronous access to the network,
- The access to the disk, especially for reading or writing voluminous files,
- The execution of complex tasks, like animations or complex data processing.

References: [4], [39]

Name: *Downloald Abuse*
Category: `Performance`
Problem: Mobile apps rely increasingly on online data storage to save their content without impacting the user's limited internal storage. Despite the positive impact of this practice on user experience, it may also lead to performance problems if used inappropriately. The *Download Abuse* code smell describes the case where the online data are downloaded with abuse, the most explicit case of this may be downloading the same data repeatedly without using the cache memory. Since the access to online data requires more energy than the access to internal disk [12], this practice negatively impacts the battery autonomy. Furthermore, depending on the network state it may also impact the execution time, and it can be costly if the user is connected to a GSM network.
References: [4], [26], [40]

C. Similarities with Android Smells

We noticed that some of the aforementioned iOS specific code smells are similar to Android code smells listed in previous works [26], [27], [57]. We focus here the Android smells that we consider as analogous to the aforementioned iOS smells with an emphasis on the commonalities.

No Low Memory Resolver (*NLMR*): when the Android system is running low on memory, the system calls the method `onLowMemory()` of every running activity. This method is responsible of trimming the memory usage of the activity. If this method is not implemented by the activity, the Android system automatically kills the process of the activity to free memory, which may lead to an unexpected program termination [27], [57]. This smell is analogous to IGNORING LOW MEMORY WARNING in iOS.

Heavy Main Thread Method (*HEAVY*): this code smell is a composition of 3 similar Android smells: *Heavy Service Start* [27], *Heavy BroadcastReceiver* [27], [28], and *Heavy AsyncTask* [27], [57]. The three code smells are defined as Android methods that contain heavy processing and are tied to the main-thread. Consequently, the three code smells lead to freezing the UI and make the app unresponsive. The main concept of the three code smells is analogous to BLOCKING THE MAIN-THREAD in iOS.

V. ADAPTING PAPRIKA FOR IOS

PAPRIKA [28] is a tooled approach that detects OO and Android smells in Android apps. It first transforms the input app into a quality model, which is stored in a database graph, and then applies queries onto these graphs to detect the occurrences of code smells. So far, PAPRIKA supports only Android apps, which are written with the Java programming language. However, iOS apps are mostly written with Objective-C and Swift. This implies that PAPRIKA requires to be extended to support these languages in order to be used as part of our study. We devote the following subsections to explain the changes we applied to PAPRIKA.

A. Code Analysis

1) From Parsing Android Apps: PAPRIKA uses the SOOT framework [74] and its DEXPLER module [10] to analyze APK artifacts. SOOT converts the Dalvik bytecode of Android apps into a SOOT internal representation, which is similar to the Java language, and also generates the call graph of the app. The representation and the graph are used by PAPRIKA to build a model of the code (including classes, methods, attributes) as a graph annotated with a set of raw quality metrics. PAPRIKA enriches this model with metadata extracted from the APK file (*e.g.*, app name, package) and the Google Play Store (*e.g.*, rating, number of downloads).

2) To Parsing iOS Apps: As SOOT does not support Objective-C nor Swift, we had to find an alternative solution. The analysis of iOS binaries is challenging due to the issues exposed in Section III, thus we opted for source code analysis. We generated two parsers for Objective-C and Swift using two ANTLR4 grammars [70], [71] and the ANTLR parser generator. Reading the source code, the parsers build an *Abstract Syntax Tree* (AST) of the analyzed program. We built a custom visitor for each parser in order to extract from the AST all the necessary attributes and metrics to feed PAPRIKA and to build the associated graph model. Since we are retrieving apps from open-source repositories (*e.g.*, GitHub) instead of the official

online stores, we cannot retrieve the complementary metadata, but this does not impact the detection process and the results we obtain.

B. Paprika Model for iOS

The PAPRIKA model built from the code analysis phase is converted into a graph where the nodes are entities, the edges are relationships, and the nodes attributes are properties or quality metrics. More details about the PAPRIKA model are given in [28]. The graph model of PAPRIKA was originally designed to reflect the core components of any Android app, and consequently it is based on the Java language elements. While Objective-C and Swift share the same core concepts of Java (classes, methods and attributes), they have also features of the procedural paradigm like functions and global variables and they introduce new concepts like extensions, structs and protocols. To cope with all these particularities, we enriched the original PAPRIKA model with new concepts and we removed few others as well. The exhaustive list of entities and metrics used for iOS is provided online[1].

C. Storing the iOS Model

PAPRIKA stores the model generated from the app analysis in a Neo4j database [47]. NEO4J is a flexible graph database that offers good performances on large-scale datasets especially when combined with the CYPHER [48] query language. We therefore reuse the storage layer of PAPRIKA as it is scalable and efficient and allows to easily store and query the graph model.

D. Code Smells Queries

PAPRIKA uses Cypher [48] queries for expressing the code smells. When applied on the graph database, these queries detect smells instances. We kept the PAPRIKA original queries for the OO smells and we just updated the metrics thresholds. The thresholds are computed using the Boxplot technique [73] by considering all the dataset apps and thus, we have different thresholds in Objective-C and Swift.
For iOS-specific smells, we follow the same process defined by PAPRIKA for transforming the literal definition of the code smell into a query. The following two examples illustrate the queries for detecting *Ignoring Low-Memory Warning* (ILMW) and *Massive View Controller* (MAVC).

Listing 1: *Ignoring Low-Memory Warning* (ILMW) query.

```
MATCH (cl:Class)
WHERE HAS(cl.is_view_controller)
  AND NOT (cl:Class)-[:CLASS_OWNS_METHOD]->
    (:Method{name:'didReceiveMemoryWarning'})
RETURN cl
```

The ILMW query looks for the view controllers missing an implementation of the method `didReceiveMemoryWarning`, which is required to react to the system memory warnings.

[1]http://sofa.uqam.ca/paprika/paprika_ios.php

Listing 2: *Massive View Controller* (MAVC) query.

```
MATCH (cl:Class)
WHERE HAS(cl.is_view_controller)
  AND cl.number_of_methods > very_high_nom
  AND cl.number_of_attributes > very_high_noa
  AND cl.number_of_lines > very_high_nol
RETURN cl
```

For the MASSIVE VIEW CONTROLLER query, classes are identified as smell instances whenever the metrics number_of_methods (`nom`), numbers_of_attributes (`noa`), and number_of_lines (`nol`) are *very high*.

It is important to note that translating smells into queries is not always possible. This can be due to the complexity of the smell itself, as for SINGLETON ABUSE, which is a very contextual smell that requires some intelligence and we cannot define an accurate rule for detecting it. Also, static analysis cannot always catch the smell concept. For example, dynamic analysis is required to measure the execution time to detect the HEBT smell and, for DOWNLOAD ABUSE, data flow analysis is needed to determine whether the downloaded data is being reused or not. Therefore, PAPRIKA cannot be used off-the-shelf to detect the smells SINGLETON ABUSE, HEBT, and DOWNLOAD ABUSE.

1) Handling the Uncertainty: Standard threshold-based approaches for computing outliers are limited as they can only report boolean values. In order to deliver results that are closer to human reasoning, PAPRIKA adopts fuzzy logic [76]. In particular, PAPRIKA uses jFuzzyLogic [14] to compute the fuzzy value between the two extreme cases of truth (0 and 1). For each metric, the *very high* value is considered as an extreme case of truth. These fuzzy values represent the degree of truth or certainty of the detected instance.

VI. STUDY OF SMELLS PRESENCE IN IOS

This section focuses on our first research question:

RQ1: Are the OO and iOS smells present in Swift and Objective-C with the same proportions?

Using our extended version of PAPRIKA, we study the presence of OO and iOS smells on a dataset of Objective-C and Swift apps. The following subsections present the details of the study, the results with a discussion, and we conclude with the related threats to validity.

A. Objects

The objects of our study are the iOS and OO smells detectable by PAPRIKA. As mentioned previously, after adapting PAPRIKA, we are able to detect three iOS smells, namely *Massive View Controller* (MAVC), *Ignoring Low Memory Warning* (ILMW), and *Blocking the Main Thread* (BTMT). In addition to these three iOS smells, PAPRIKA can detect four well-known OO smells, namely BLOB, *Swiss Army Knife* (SAK), *Long Method* (LM), and *Complex Class* (CC) [11], [24].

B. iOS Dataset and Inclusion Criteria

Following the technical choices we adopted to extend PAPRIKA for iOS, we need the source code of the apps to analyze, thus our dataset is exclusively composed of open-source apps. We chose to consider a collaborative list of apps available from Github [21] that is, as far as we know, the largest repository of open-source iOS apps available online. It gathers currently 605 apps of diverse categories and developed by different developers, almost $47\,\%$ of them being published in the App Store.

These apps are written with Objective-C, Swift, Xamarin, and other web languages such as HTML and JavaScript. We considered only the native apps written with Objective-C and Swift. We also excluded demo and tutorial apps since they are very light and not relevant for our study. As a result we included 103 Objective-C apps and 176 Swift apps within this study. The exhaustive list of apps used in our study is available online.[2]

C. Hypotheses and Variables

Independent variable: The independent variable of this study is the programming language of the app: Objective-C (*ObjC*) or *Swift*.

Dependent variables: The dependent variables are the proportions of code smells in the analyzed apps.

Hypothesis: To compare the presence of the different code smells ($CS \in \{Blob, LM, CC, SAK, MaVC, ILMW, BTMT\}$) in the two languages ($L \in \{ObjC, Swift\}$), we formulate the following null hypothesis, which we applied to the two datasets:

HR^{CS}:There is no difference between the *proportions* of code smells for the apps written with Objective-C or Swift;

D. Analysis Method

First, we analyze the Objective-C and Swift datasets to identify the OO and iOS smells. Then, we compute, for each app, the ratio between the number of code smells and the number of concerned entities—*e.g.* the ratio of ILMW is normalized with the number of view controllers. Afterwards, we compute the median and the interquartile range of the ratios. We also compute Cliff's δ [59] effect size to quantify the importance of the difference in proportions of the smells.

Cliff's δ is reported to be more robust and reliable than Cohen's d [17]. It is a non-parametric effect sizes measure—*i.e.*, it makes no assumptions of a particular distribution—which represents the degree of overlap between two sample distributions [59]. It ranges from -1 (if all the selected values in the first group are larger than the ones of the second group) to $+1$ (if all the selected values in the first group are smaller than the second group). It evaluates to zero when two sample distributions are identical [16]:

$$\text{Cliff's } \delta = \begin{cases} +1, & \text{Group 1} > \text{Group 2;} \\ -1, & \text{Group 1} < \text{Group 2;} \\ 0, & \text{Group 1 = Group 2.} \end{cases}$$

Interpreting the effect sizes: Cohen's d is mapped to Cliff's δ via the percentage of non-overlap, as shown in Table I [59]. Cohen [18] states that a medium effect size represents a difference likely to be visible to a careful observer, while a large effect is noticeably larger than medium.

TABLE I: 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

We chose to use the median and the interquartile range because our data are not normally distributed. Likewise, we opted for the Cliff's δ test since it is suitable for non-normal distributions. Moreover, Cliff's δ is also recommended for comparing samples of different sizes [36].

E. Results

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

1) Overview of the Results: Table II synthesizes the percentages of apps affected by each code smell for the two datasets.

TABLE II: Percentage of apps affected by smells.

	Lang	BLOB	LM	CC	SAK	MAVC	ILMW	BTMT
Apps %	$ObjC$	58.82	96.08	76.47	24.51	33.33	85.29	6.86
	$Swift$	40.34	87.50	69.32	14.77	10.23	78.41	0.00

For the iOS smells, we observe that ILMW appears in most of the apps. The method `applicationDidReceiveMemoryWarning:` is not implemented in most of the cases. Missing the implementation of this method can be justified when absolutely no resource can be freed by the app, however it is unlikely that such a large proportion of activities belongs to this case. We therefore hypothesize that developers are not aware of the benefits of implementing this method.

MAVC is also relatively common, as it appears in more than $10\,\%$ of the apps in the two datasets. Though, BTMT is not a recurrent code smell, as it appears in only $6\,\%$ of the Objective-C apps, and it does not appear at all in the Swift ones.

Generally, we can observe that iOS smells tend to be more present in the Objective-C apps than the Swift ones. Nonetheless, this insight needs to be consolidated in order to conclude before the comparative study of the next subsection.

Regarding the OO smells, LM is the most recurrent one, as it appears in approximately $90\,\%$ of the apps of the two datasets. BLOB and CC are also very common with percentages ranging from $40\,\%$ to $76\,\%$. SAK is less prevalent, but its percentage is still significant. Similarly to the iOS smells, the percentages show that Objective-C apps are more prone to OO smells than Swift apps.

[2] http://sofa.uqam.ca/paprika/paprika_ios.php

2) Comparison between Objective-C and Swift: The following table shows, for each smell and programming language, the median (med) and interquartile range (IQR) of the ratios of smells in the apps. Also, it reports on the Cliff's δ effect size between the ratios in the two datasets.

For each application a, the ratio of the smell s is defined by:

$$ratio_s(a) = \frac{fuzzy\_value_s(a)}{number\_of\_entities_s(a)}$$

where $fuzzy\_value_s(a)$ is the sum of the fuzzy values of the detected instances of the smell s in the app a and $number\_of\_entities_s(a)$ is the number of the entities concerned by the smell s in the app a. As a reminder, for BLOB, CC and BTMT, the concerned entity is the *class*, while for LM it is the *method*. For MAVC and ILMW, it is the *view controller* and for SAK it is the *interface*.

The Cliff's δ values are evaluated according to the mapping defined in Table I. If a value is under the SMALL range, the difference is considered INSIGNIFICANT.

TABLE III: Ratios comparison between Objective-C and Swift.

Smell	Lang	Med	IQR	Cliff's δ
BLOB	$ObjC$	0.004	0.020	0.304(S)
	$Swift$	0.000	0.004	
LM	$ObjC$	0.060	0.055	0.135(I)
	$Swift$	0.048	0.059	
CC	$ObjC$	0.033	0.071	0.062(I)
	$Swift$	0.026	0.074	
SAK	$ObjC$	0.000	0.000	0.115(I)
	$Swift$	0.000	0.000	
MAVC	$ObjC$	0.000	0.015	0.181(S)
	$Swift$	0.000	0.000	
ILMW	$ObjC$	0.905	0.634	0.156(S)
	$Swift$	0.583	0.833	
BTMT	$ObjC$	0.000	0.000	0.069(I)
	$Swift$	0.000	0.000	

I: INSIGNIFICANT DIFFERENCE.
S: SMALL DIFFERENCE.

First, we observe that the median ratio for SAK, MAVC and BTMT is null. This is consistent with the results of Table II, since these smells appear in less than $35\,\%$ of the apps, which means that in more than $70\,\%$ of the apps the ratio is null. Consequently, the first quartile and the median for their ratios are null and the IQR is very small or almost null.

We also observe that the IQR is very small for all the smells except ILMW. These weak values show that these smells are present in the apps with close ratios. As for ILMW, the high IQR value indicates that this smell is abundant in some apps, but nearly absent in the others, and this enforces our hypothesis in the previous subsection about the ILMW origin. In other words, the majority of the developers are not aware of the memory warnings importance and the ILMW ratio is high in their apps, while few other developers are aware of the issue and they have much less ILMW instances in their apps.

The cliff's δ values show that the difference between the smell ratios in Objective-C and Swift is insignificant for LM, CC, SAK, and BTMT. Moreover, the median and the IQR for these smells are very close in the two datasets. Therefore, we deduct that these smells are present in the two datasets with the same proportions and we can accept the study hypothesis HR^{CS} for $CS \in \{LM, CC, SAK, BTMT\}$.

For BLOB, MAVC and ILMW, there are *small* differences in the smell ratios in the two languages. Thus, we reject the hypothesis HR^{CS} for $CS \in \{BLOB, MaVC, ILMW\}$, and we conclude that these smells are slightly more present in Objective-C than Swift.

The similar proportions of ILMW, MAVC and BTMT between Objective-C and Swift are quite expectable, as these smells are related to the platform concepts, which are shared by the two datasets apps. Concerning OO smells, we were expecting different proportions of code smells as the metrics used to detect these smells (*e.g.*, complexity, number of lines) are dependent of the language structure and features. Thus, we expected concepts, such as protocols, generics or functional programming, which are presents in Swift but not in Objective-C to affect the proportion of code smells. However, we have similar proportion in our results. To further investigate this point, we look at the metrics related to these smells and we compare them. For this purpose, we compute for each metric the median, first and third quartile, and the standard deviation. The quartiles allow us to observe how the metrics are distributed, and the standard deviation to catch the effect of outliers. We also use the Wilcoxon–Mann–Whitney test [61] to check if the distributions of our metrics in both datasets are identical, so that there is a $50\,\%$ probability that an observation from a value randomly selected from the Objective-C dataset will be identical to an observation randomly selected from the Swift dataset. To perform this comparison, we use a $99\,\%$ confidence level—*i.e.*, p-value < 0.01. Table IV depicts the computed values for the metrics: *class complexity* (CLC), *number of attributes* (NOA), *number of methods* (NOM), *number of lines in a method* (NOL_M), *number of lines in a class* (NOL_C), and *number of methods in an interface* (NOM_I).

For the Wilcoxon–Mann–Whitney test, since the p-values for all the metrics are less than the threshold ($p = 0.01$), we cannot accept the hypothesis of the similarity in the two distributions. Moreover, the results indicate that the quartile values in Objective-C are considerably higher than Swift, with also a much higher standard deviation. This means that the metrics range is wider, but also that there are more outliers in Objective-C.

These results clearly show that the apps written in Objective-C and Swift are very different in terms of OO metrics. Compared to Swift, Objective-C apps tend to have longer and more complex methods and classes, with more attributes and methods in classes and interfaces. This is likely due to the features introduced in Swift that help the developer to better structure the apps. As an example, the *Extensions* to existing types, the *Structs*, and the advanced *Enums* are

TABLE IV: Metrics comparison between Objective-C and Swift.

Metric	Lang	Q1	MED	Q3	SD	p-value
CLC	$ObjC$	2	6	16	22.34	0
	$Swift$	0	2	5	11.72	
NOA	$ObjC$	0	3	6	6.84	0
	$Swift$	0	0	1	2.58	
NOM	$ObjC$	1	4	8	8.57	0
	$Swift$	0	1	3	8.41	
NOL_M	$ObjC$	3	6	13	20.70	1.7e-39
	$Swift$	2	5	11	26.09	
NOL_C	$ObjC$	12	39	108	153.84	2.6e-196
	$Swift$	5	17	43	143.19	
NOM_I	$ObjC$	1	1	3	2.87	1.1e-227
	$Swift$	0	1	2	2.61	

features that encourage developers to implement lighter classes in Swift.

Following this deduction, we can affirm that the presence of OO smells with the same proportions in iOS apps is not due to the similarity between the languages Objective-C and Swift. We rather hypothesize that it is due to the development framework, as it dictates the architecture and some practices, as well as the developers habits. In fact, the developers community is almost the same for the two languages.

F. Threats to Validity

We discuss here the main issues that may have threatened the validity of the validation study, by considering the classification of threats proposed in [20]:

Internal Validity: The threats to internal validity are relevant in those studies that attempt to establish causal relationships. In this case, the internal validity might be affected by the detection strategy of PAPRIKA. We rely on a robust set of standard metrics to evaluate the presence of a well-defined set of code smells in the analyzed applications. However, for the threshold-based code smells, we computed the thresholds from the analysis of the whole dataset, to avoid influencing the detection results.

External Validity: This refers to the approximate truth of conclusions involving generalizations within different contexts. The main threat to external validity is the representativeness of the results. We used sets of 103 and 176 representative open-source apps. It would have been preferable to consider non open-source apps to build a bigger and more diverse dataset. However, this option was not possible as the analysis of iOS binaries is restricted. We believe that our dataset still allows us to generalize our results to open-source iOS apps, but that further studies are needed to extend our results to all iOS apps.

Construct Validity: The threats to construct validity concern the relation between theory and observations. In this study, these threats could be due to errors during the analysis process. We were very careful when collecting and presenting the results and drawing conclusions based on these results.

Conclusion Validity: This threat refers to the relation between the treatment and the outcome. 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 [43]. We paid attention not to make conclusions that cannot be validated with the presented results.

VII. COMPARATIVE STUDY BETWEEN IOS AND ANDROID

In this section, we conduct a comparative study between iOS and Android apps to observe how code smells proportions may differ in the two platforms. A previous study [37] already explored the difference of the proportion of code smells between Android and Java desktop applications. Our goal here is to compare the two mobile-specific platforms in a similar way, by answering our second research question :

RQ2: Is there a difference between iOS and Android in terms of code smells presence?

A. Android Dataset and Inclusion Criteria

In order to compare iOS and Android apps, we use the iOS dataset described in Section VI-E and an Android dataset composed of $1,551$ open-source Android apps. We consider $1,551$ Android apps that differ both in internal attributes, such as their size, and external attributes from the perspective of end users, such as user ranking and number of downloads. These apps were collected from the F-Droid repository[3] in April 2016. The dataset contains all the apps available at this time that we could analyze with PAPRIKA. F-Droid is a non-profit volunteer project, which aims to collect all kinds of Android open-source apps. The detailed list of apps included in the dataset can be found online.[4]

B. Variables and Hypotheses

Independent variables: The nature of the app, Android or iOS, is the independent variable of our study.

Dependent variable: The dependent variables correspond to:

- the average proportion of the OO smells: BLOB, LM, CC, SAK,
- the average proportion of the similar iOS and Android smells: NLMR/ILMW, HEAVY/BTMT.

Hypotheses: To answer our second research question, we formulate the following null hypothesis, which we applied to the iOS and Android datasets ($CS \in \{Blob, LM, CC, SAK, NLMR/ILMW, HEAVY/BTMT\}$):

HR^{CS}:There is no difference between the *proportions* code smells in Android and iOS apps;

[3]F-Droid: https://f-droid.org

[4]http://sofa.uqam.ca/paprika/paprika_ios.php

C. Analysis Method

To compare our results, we use the median and the interquartile range. We also compute the Cliff's δ effect size to quantify the importance of the difference of the platform on the smells presence. Later, to discuss and explain the obtained results we compare the metrics values in the two platforms using the Wilcoxon and Mann-Whitney test.

As mentioned previously, Cliff's δ and Mann-Whitney make no assumption about the assessed variables distribution, and both of them are suitable for comparing datasets of different sizes.

D. Results

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

TABLE V: Percentage of Android apps affected by smells.

	BLOB	LM	CC	SAK	HEAVY	NLMR
Apps %	78.40	98.19	86.59	12.31	41.84	98.32

1) Overview of the Results for Android: We can observe in Table V that BLOB, LM and CC appear in most of the apps (more than 75% in all cases). Indeed, most apps tend to contain at least one of these code smells. On the other side, just like in iOS, SAK is uncommon and only appears in 12% of the analyzed apps. Concerning Android-specific code smells, the HEAVY code smells appear in almost half of the apps. After inspection, we observed that usually there is only one instance of these smells for each app. NLMR is our most common smell with more than 98% of the apps concerned. Here, we also noticed that there is often only one activity affected per app.

2) Comparison of Code Smells Proportions between iOS and Android: For the comparison of proportions of code smells between iOS and Android, we compare the ratio between the number of code smells and the number of concerned entities for each code smell in every app. The medians, IQR and Cliff's δ effect sizes of the distributions obtained are presented in Table VI. The Cliff's δ values column presents for each smell the Cliff test for Android with Objective-C, then for Android and Swift, respectively.

At first glance, we can observe that we have a significant difference between the proportions of code smells in Android compared to Objective-C and Swift in all the cases, except SAK. For OO smells, Cliff's δ always gives a large difference for BLOB, CC and LM. Furthermore, this is confirmed by the greater means in Android apps for these smells and thus, we reject the hypothesis HR^{CS} for all OO code smells, except SAK.

We accept HR^{SAK} since Cliff's δ shows no significant difference and its median and IQR are almost the same in the two platforms. As explained before, SAK is uncommon in both Android and iOS apps.

Concerning similar mobile smells between Android and iOS, we reject the hypothesis HR^{NLMR}. In the case of Objective-C, there is a medium difference while there is a large difference for Swift. In both cases, Android apps have a bigger proportion of code smells as NLMR appears almost in all Android apps (cf. Table V). On the contrary, there are many apps without ILMW on iOS. This is confirmed by the median at 1 and the IQR at 0, while the median is under 1 with a non-null IQR for iOS apps.

TABLE VI: Ratios comparison between Android and iOS

Smell	Lang	Med	IQR	Cliff's δ
BLOB	$ObjC$	0.004	0.020	0.495(L)
	$Android$	0.052	0.129	0.648(L)
	$Swift$	0.000	0.004	
LM	$ObjC$	0.060	0.055	0.710(L)
	$Android$	0.156	0.289	0.727(L)
	$Swift$	0.048	0.059	
CC	$ObjC$	0.033	0.071	0.509(L)
	$Android$	0.122	0.317	0.518(L)
	$Swift$	0.026	0.074	
SAK	$ObjC$	0.000	0.000	-0.121(I)
	$Android$	0.000	0.000	-0.015(I)
	$Swift$	0.000	0.000	
NLMR/ILMW	$ObjC$	0.905	0.634	0.397(M)
	$Android$	1.000	0.000	0.512(L)
	$Swift$	0.583	0.833	
HEAVY/BTMT	$ObjC$	0.000	0.000	0.261(S)
	$Android$	0.000	0.007	0.331(M)
	$Swift$	0.000	0.000	

I: INSIGNIFICANT DIFFERENCE.
S: SMALL DIFFERENCE.
M: MEDIUM DIFFERENCE.
L: LARGE DIFFERENCE.

We also reject the hypothesis HR^{HEAVY}. In the case of Objective-C, there is a small difference while there is a medium difference for Swift. Again, Android apps have a bigger proportion of HEAVY code smells, but the medians remains at 0, which means than most apps do not contain these smells. However, Android apps have a bigger IQR with still a significant proportion of apps containing these smells while the BTMT is uncommon in Objective-C and Swift.

In summary, Android apps tend to contain more code smells than iOS apps in both languages at the exception of the SAK code smells, which appear in the same proportion for all languages. It should be noted that, while the results clearly indicate that the Android apps have more outliers than the iOS ones, this does not necessarily mean that Android apps are globally worst in terms of quality. To further examine the differences between the two platforms apps, we consider the metrics values.

We compare the distributions of common metrics that we use for our request to detect iOS and Android code smells. We excluded the metrics of the number of lines in methods and classes because in Android PAPRIKA uses the number of instructions. The number of instructions does not always correspond to the number of lines, and thus we cannot compare

the two metrics.

TABLE VII: Metrics comparison between iOS and Android.

Metric	Lang	Q1	MED	Q3	SD	p-value
CLC	$ObjC$	2	6	16	22.34	0.71
	$Android$	3	5	13	35.92	
	$Swift$	0	2	5	11.72	0
NOA	$ObjC$	0	3	6	6.84	1e-48
	$Android$	1	2	4	22.25	
	$Swift$	0	0	1	2.58	0
NOM	$ObjC$	1	4	8	8.57	1.57-11
	$Android$	2	4	7	12.61	
	$Swift$	0	1	3	8.41	0
NOM_I	$ObjC$	1	1	3	2.87	0.26
	$Android$	1	1	3	6.97	
	$Swift$	0	1	2	2.61	2.9e-140

The p-values show a disparity between the metrics values in the different datasets. Except for the complexity, the metrics in Android are different for both Objective-C and Swift. Objective-C has the highest values in the median and the third quartile, as explained before this is probably due to the language nature. Android has also high metrics values, but less than Objective-C.

These results support our hypothesis about the Android apps quality. The Android apps do not have the highest metric values but they show the highest standard deviation, which means that they tend to have more outliers. These outliers may be attributed to the platform specificities, like the architecture and framework constraints, or to the lack of documentation and code quality assessment tools. Future works can involve developers in this study to further investigate this point.

E. Threats to Validity

In this subsection, we analyze the factors that may threaten the validity of this study using the same classification applied in Section VI-F:

Internal Validity: Again, in this case, the internal validity might be affected by the detection strategy of PAPRIKA. For the Android dataset, we also tried to rely on a robust set of metrics to evaluate the presence of a well-defined set of code smells in the analyzed applications. However, for the threshold-based code smells, we calculated the thresholds based on the analysis of the whole dataset, so as not to influence the detection results.

External Validity: In this study, we analyzed a large, heterogeneous dataset of Android open-source apps available on F-Droid. In a similar way, we tried to use as many open-source iOS apps as possible, as mentioned in the previous study. Despite our efforts, the iOS datasets are significantly smaller and thus probably less representative than our Android dataset. However, our analysis remains meaningful as we were careful to only conclude on statistical tests, which are suitable for samples of different sizes. We believe that all datasets provide a neutral and diverse set of apps, but it should also be noted that, in this study, we are focusing only on open-source apps and thus, we cannot generalize our results to all available apps. Further investigations are needed to confirm that open-source and non open-source apps are similar in terms of code smells.

Construct Validity: The construct validity can be threatened by the way we are computing thresholds. Indeed, we could use the same thresholds for all detections (*e.g.*, by using an average threshold between Android and iOS). However we believe that in this case the thresholds will not be relevant regarding the contexts. Moreover, boxplots are commonly used for assessing values in empirical studies [43].

Conclusion Validity: As before, the main threat to the conclusion validity in this study is the validity of the statistical tests applied, we alleviated this threat by applying the same tests.

VIII. CONCLUSION AND FUTURE WORKS

In this paper, we conducted a study related to the presence of code smells in the iOS mobile platform. We proposed a catalog of 6 smells that we identified from the documentation and the developers feedbacks, these smells are specific to iOS and have not been reported in the literature before. We also presented an adaptation of the PAPRIKA tooled approach, which was originally designed for detecting smells in Android apps, to detect smells in iOS apps written with Objective-C and Swift.

Based on the smells catalog and the extended PAPRIKA toolkit, we performed two studies to investigate the scope of the code smells in mobile apps. We first analyzed 279 iOS apps with PAPRIKA and compared the proportions of smells in apps written with Objective-c and Swift. We observed that the Objective-C and Swift apps are very different in terms of quality metrics. In particular, the Swift apps tend to be lighter. However, these differences do not seem to impact the smells proportions since the two types of apps exhibit OO and iOS smells equally. Then, we compared iOS apps with Android apps in order to investigate the potential differences between the two platforms. The comparison showed that Android apps tend to have more smells than the iOS ones, and again we demonstrated that this is not due to the programming language, but rather potentially to the platform.

This work provided interesting insights for both developers and scientific community. It highlighted some of the common smells used in the iOS apps and provided relevant information about the code quality in a mobile platform, which has not been addressed by the community before. Moreover, we provided an open-source toolkit, PAPRIKA, which is—to the best of our knowledge—the first to support the detection of smells in iOS apps. This facilitates and encourages conducting further studies on iOS apps.

In future works, we are planning to involve developers in this study in order to explain the reasons for the smells introduction. We are also planning to include iOS binaries analysis to take advantage of the huge number of apps available in the online stores. This would allow us to work on bigger datasets and have more generalizable results.

REFERENCES

[1] Agiletribe. Don't abuse singleton pattern. https://agiletribe.wordpress.com/2013/10/08/dont-abuse-singleton-pattern/, 2013. accessed: 2017-01-06.

[2] Android. Android Lint - Android Tools Project Site. http://tools.android.com/tips/lint, 2015. accessed: 2017-01-20.

[3] Apple. Threading programming guide. https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/Multithreading/Introduction/Introduction.html, 2014. accessed: 2017-01-06.

[4] Apple. Performance tips. https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/PerformanceTips/PerformanceTips.html, 2015. accessed: 2017-01-06.

[5] Apple. The Singleton pattern in Cocoa. https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/Singleton.html#//apple_ref/doc/uid/TP40008195-CH49-SW1, 2015. accessed: 2016-10-20.

[6] Apple. Work less in the background. https://developer.apple.com/library/ios/documentation/Performance/Conceptual/EnergyGuide-iOS/WorkLessInTheBackground.html#//apple_ref/doc/uid/TP40015243-CH22-SW1, 2015. accessed: 2016-10-20.

[7] Apple. Apple developer documentation. https://developer.apple.com/develop/, 2017. accessed: 2017-01-09.

[8] F. Ash. Introduction to mvvm. https://www.objc.io/issues/13-architecture/mvvm/, 2014. accessed: 2017-01-06.

[9] Atom. Linter swiftc. https://atom.io/packages/linter-swiftc, 2016. accessed: 2017-01-09.

[10] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proc. of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38. ACM, 2012.

[11] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1998.

[12] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 21–21, 2010.

[13] E. Chris. Lighter view controllers. https://www.objc.io/issues/1-view-controllers/lighter-view-controllers/, 2013. accessed: 2017-01-06.

[14] P. Cingolani and J. Alcalá-Fdez. jfuzzylogic: a java library to design fuzzy logic controllers according to the standard for fuzzy control programming. *International Journal of Computational Intelligence Systems*, 6(sup1):61–75, 2013.

[15] Clang. Clang static analyzer. http://clang-analyzer.llvm.org/, 2016. accessed:2016-10-20.

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

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

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

[19] W. Colin. Singletons: You're doing them wrong. http://cocoasamurai.blogspot.fr/2011/04/singletons-your-doing-them-wrong.html, 2011. accessed: 2017-01-06.

[20] T. Cook and D. Campbell. *Quasi-experimentation: Design & analysis issues for field settings*. Houghton Mifflin Company, 1979.

[21] Crowd. Collaborative list of open-source ios apps. https://github.com/dkhamsing/open-source-ios-apps, 2016. accessed: 2016-10-20.

[22] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *NDSS*, 2011.

[23] FauxPas. Faux pas. http://fauxpasapp.com/#highlights, 2016. accessed: 2017-01-09.

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

[25] M. Gjoshevski and T. Schweighofer. Small Scale Analysis of Source Code Quality with regard to Native Android Mobile Applications. In *4th Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications*, pages 2–10, Maribor, Slovenia, 2015. CEUR Vol. 1375.

[26] M. Gottschalk, J. Jelschen, and A. Winter. Saving Energy on Mobile Devices by Refactoring. *28th International Conference on Informatics for Environmental Protection (EnviroInfo 2014)*, 2014.

[27] G. Hecht. *Detection and analysis of impact of code smells in mobile applications*. Theses, Université Lille 1 : Sciences et Technologies ; Université du Québec à Montréal, Nov. 2016.

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

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

[30] M. Hector. Defeating the antipattern bully. https://krakendev.io/blog/antipatterns-singletons, 2015. accessed: 2017-01-06.

[31] Infer. Infer. http://fbinfer.com/, 2016. accessed:2016-04-29.

[32] G. Jeff and S. Conrad. Architecting ios apps with viper. https://www.objc.io/issues/13-architecture/viper/, 2014. accessed: 2016-10-20.

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

[34] C. Kyle. Why singletons are bad. http://www.kyleclegg.com/blog/9272013why-singletons-are-bad, 2013. accessed: 2017-01-06.

[35] 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.

[36] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma. Cliff's delta calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica*, 10(2):545–555, 2011.

[37] 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.

[38] F. Marcelo. 25 ios app performance tips and tricks. http://www.raywenderlich.com/31166/25-ios-app-performance-tips-tricks#memwarnings, 2013. accessed: 2017-01-06.

[39] F. Marcelo. 25 ios app performance tips and tricks. https://www.raywenderlich.com/31166/25-ios-app-performance-tips-tricks#mainthread, 2013. accessed: 2017-10-06.

[40] F. Marcelo. 25 ios app performance tips and tricks. https://www.raywenderlich.com/31166/25-ios-app-performance-tips-tricks#cache, 2013. accessed: 2017-01-06.

[41] Z. Marcus. Massive view controller. http://www.cimgf.com/2015/09/21/massive-view-controllers/, 2015. accessed: 2017-01-06.

[42] M. Matteo. How to keep your view controllers small for a better code base. http://matteomanferdini.com/how-to-keep-your-view-controllers-small-for-a-better-code-base/, 2014. accessed: 2017-01-06.

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

[44] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[45] 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.

[46] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. 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.

[47] Neo4j. NEO4J. http://neo4j.com. accessed: 2017-01-09.

[48] Neo4j. Cypher query language. http://neo4j.com/docs/stable/cypher-query-lang.html, 2016. accessed: 2016-05-17.

[49] Objc.io. Objc.io. https://www.objc.io/, 2017. accessed: 2017-01-09.

[50] OCLint. Oclint. http://oclint.org/, 2016. accessed:2016-04-29.

[51] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 390–400, Washington, DC, USA, 2009. IEEE Computer Society.

[52] OpenTheFile. What is an ipa file and how do i open an ipa file? http://www.openthefile.net/extension/ipa, 2015. accessed: 2016-05-17.

[53] Oracle. Java platform, micro edition (java me). http://www.oracle.com/technetwork/java/embedded/javame/index.html. accessed: 2017-01-09.

[54] L. Raymond. Clean swift ios architecture for fixing massive view controller. http://clean-swift.com/clean-swift-ios-architecture/, 2015. accessed: 2017-01-06.

[55] Raywenderlich. Raywenderlich. https://www.raywenderlich.com/, 2016. accessed: 2017-01-09.

[56] Realm. Swift lint. https://github.com/realm/SwiftLint, 2016. accessed: 2017-01-09.

[57] 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.

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

[59] 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.

[60] 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.

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

[62] SonarQube. Sonarqube. http://www.sonarqube.org/, 2015. accessed: 2016-10-20.

[63] K. Soroush. Massive view controller. http://khanlou.com/2015/12/massive-view-controller/, 2015. accessed: 2017-01-06.

[64] SourceForge. Pmd. https://pmd.github.io/, 2016. accessed: 2016-05-30.

[65] stackoverflow. Stack overflow. http://stackoverflow.com/, 2017. accessed: 2017-01-09.

[66] P. Stephen. Avoiding singleton abuse. https://www.objc.io/issues/13-architecture/singletons/#avoiding-singletons, 2014. accessed: 2017-01-06.

[67] A. Strauss and J. Corbin. Grounded theory methodology: An overview. In N. K. Denzin and Y. S. Lincoln, editors, *Handbook of Qualitative Research*, pages 273–285, Thousand Oaks, CA, 1994. Sage Publications.

[68] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna. Challenges for dynamic analysis of ios applications. In *Open Problems in Network Security*, pages 65–77. Springer, 2012.

[69] Tailor. Tailor swift static analyzer. https://tailor.sh/, 2016. accessed: 2016-10-20.

[70] P. Terence. Objc.g4. https://github.com/antlr/grammars-v4/tree/master/objc, 2015. accessed:2017-01-06.

[71] P. Terence. Swift.g4. https://github.com/antlr/grammars-v4/blob/master/swift/Swift.g4, 2016. accessed:2017-01-06.

[72] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 329–331. IEEE, 2008.

[73] J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[74] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proc. of the conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

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

[76] L. A. Zadeh. Fuzzy logic and its application to approximate reasoning. In *IFIP Congress*, pages 591–594, 1974.