Understanding Code Smells in Android Applications

Umme Ayda Mannan
Oregon State University
Corvallis, OR, USA
mannanu@oregonstate.edu

Iftekhar Ahmed
Oregon State University
Corvallis, OR, USA
ahmedi@oregonstate.edu

Rana Abdullah M Almurshed
Oregon State University
Corvallis, OR, USA
almurshr@oregonstate.edu

Danny Dig
Oregon State University
Corvallis, OR, USA
digd@eecs.oregonstate.edu

Carlos Jensen
Oregon State University
Corvallis, OR, USA
cjensen@eecs.oregonstate.edu

ABSTRACT

Code smells are associated with poor coding practices that cause long-term maintainability problems and mask bugs. Despite mobile being a fast growing software sector, code smells in mobile applications have been understudied. We do not know how code smells in mobile applications compare to those in desktop applications, and how code smells are affecting the design of mobile applications. Without such knowledge, application developers, tool builders, and researchers cannot improve the practice and state of the art of mobile development.

We first reviewed the literature on code smells in Android applications and found that there is a significant gap between the most studied code smells in literature and most frequently occurring code smells in real world applications. Inspired by this finding, we conducted a large scale empirical study to compare the type, density, and distribution of code smells in mobile vs. desktop applications. We analyze an open-source corpus of 500 Android applications (total of 6.7M LOC) and 750 desktop Java applications (total of 16M LOC), and compare 14,553 instances of code smells in Android applications to 117,557 instances of code smells in desktop applications. We find that, despite mobile applications having different structure and workflow than desktop applications, the variety and density of code smells is similar. However, the distribution of code smells is different – some code smells occur more frequently in mobile applications. We also found that different categories of Android applications have different code smell distributions. We highlight several implications of our study for application developers, tool builders, and researchers.

1. INTRODUCTION

Code smells [12] identify bad design or coding practices. Code smells are not the same as bugs and do not mean that the code deviates from the expected execution, rather that design rules were violated, which may lead to long-term maintainability problems and technical debt [5, 34]. Researchers have shown that a large number of code smells correlate with bugs [23, 30] and maintainability problems [12]. However, according to Yamashita et al. [42], around 32% of the developers are not aware of code smells and their pitfalls. Moreover, on the research side, Ahmed et al. [2] found that there is significant gap between the code smells that receive a lot of attention in the literature and those that appear most frequently in real-world applications.

In recent years, mobile applications have grown to become a large part of the software industry. According to Gartner[1], in 2016 more than 300 billion apps will be downloaded. Another Gartner[2] report shows that in 2015, Android had more than 78% of the world market share of smartphones. Thus, in this paper we focus on Android applications.

Researchers [40] have shown that code accrues code smells during high-intensity, frequent code changes performed under time and market demand. Android applications often have more frequent updates and releases than desktop applications [28]. Does this mean that Android applications exhibit more or different code smells than desktop applications?

Moreover, there are other important differences between mobile and desktop applications[3]. Mobile applications have limited resources (e.g., memory, CPU, network, battery). The programming paradigm is also different: mobile applications use reactive, event-driven programming. Android applications have a special structure: there is no main function, the entry points are given by event-handlers[4] such as onCreate, onResume, etc. Also, the libraries are different: Android does not have all J2SE APIs, nor Swing, nor JavaFX. Many APIs are specific to mobile (Contacts, Power Management, Graphics, etc.). GUIs on Android are declared via XML. Do these differences in structure and workflow of Android applications affect the distribution of code smells?

If code smells are the same, it means that all the tools and research on code smells of desktop application applies to mobile applications. But if they are different, then application developers, tool builders, and researchers can make wrong assumptions about code smells in mobile applications. Novel tools and approaches might be needed, and the priority in developing software engineering tools might need to be revised.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MOBILESoft'16, May 16–17, 2016, Austin, TX, USA.
Copyright 2016 ACM. ISBN 978-1-4503-4178-3…$15.00.
DOI: http://dx.doi.org/10.1145/12345.67890

[1] http://www.gartner.com/newsroom/id/2153215

[2] http://www.gartner.com/newsroom/id/3061917

[3] http://gamedev.stackexchange.com/questions/4288/how-different-is-java-for-jre-vs-java-for-android

[4] http://developer.android.com/guide/components/activities. html

Researchers [13, 14, 15, 31, 33, 40, 41] have just started to analyze anti-patterns and code smells in the context of Android applications. However, we did not find any systematic literature review on code smells in Android projects. We did a literature survey of the top Software Engineering conference papers from 2008 to 2015 (Section 3). Surprisingly we found only 5 studies [14, 15, 33, 40, 41] that specifically discussed code smells in Android applications. Most of these studies analyzed fewer than 50 Android applications for a set of 3 to 5 kinds of code smells. To the best of our knowledge, the previous work did not study how code smells in mobile applications compare to those in desktop applications.

In this paper we shed light on the differences between code smells in mobile applications and those in desktop applications. To minimize confounding factors, we keep several criteria constant across the two domains: the development process and culture (open-source), the programming language (Java), and the host infrastructure (GitHub). Our corpus contains 500 Android applications and 750 desktop Java applications. After downloading these projects, we ran a state-of-the-art code smell detection tool, inFusion [19], which can identify 22 different kinds of code smells.

Using our rich corpus, we answer the following research questions:

RQ1: What is the variety, density, and distribution of code smells in Android applications? How does it compare with desktop applications? Surprisingly, despite mobile applications being more frequently changed and released than desktop applications, the density of code smells is the same. Also, despite mobile applications having different structure, the variety of code smells is the same in mobile and desktop applications. However, we found that the distribution of code smells is different: in mobile applications the most frequent smells are Data Class [12] and Data Clumps [12], whereas in desktop applications the most frequent smells are External Duplication [12] and Internal Duplication [12].

RQ2: What is the distribution of code smells across categories of mobile apps (e.g., development, home, education, etc.)? We found that some categories are more prone to code smells. For example, applications under home and education category are more prone to the Data Class code smell than communication applications. However, these results were not statistically significant.

Our study has several implications for several audiences. Beyond the general guideline that developers should pay attention to code smells, our results help developers become aware of code smells that appear more prevalently in mobile applications.

The good news for tool builders is that code smell detection tools for desktop software can also be effectively reused in mobile applications. Lastly, researchers and tool builders can learn from our study about code smells that appear frequently in real-world applications, and be less biased in their selection. Thus, they can focus on improving tool support for detecting and fixing the code smells for example through automated refactoring.

This paper makes the following contributions:

1. **Problem statement:** To the best of our knowledge, we present the first study that compares code smells in mobile vs. desktop applications.
2. **Literature Survey:** We surveyed the papers in the top SE conferences for the last 8 years. We found a significant gap between the most studied code smells in literature and those that appear in real world applications.
3. **Corpus:** In contrast to previous studies [13,14,15,31,33,40,41] on code smells in mobile applications, ours is the largest study so far in terms of the number of projects analyzed (750 open-source desktop Java projects and 500 Android projects) and the kinds of analyzed code smells (22).
4. **Implications:** We present implications of our study for application developers, tool builders, and researchers.

The remainder of the paper is organized as follows: in Section 2 we discuss the definition of key code smells. Section 3 presents our literature survey results. Section 4 describes our methodology, filtering criteria, the demographics of our corpus, as well as the tool selection and data collection process. Section 5 describes the results of our study. Section 6 describes the threats to validity. In Section 7 we describe the implication of our study, and in section 8 reviews the research on code smells. Section 9 concludes with a summary of the key findings and future work.

2. A GENTLE INTRODUCTION TO CODE SMELLS

According to Martin Fowler, a code smell is a *"surface indication that usually corresponds to a deeper problem in the system"*. The term "code smell" was first introduced by Kent Beck and Martin Fowler in their seminal refactoring book, "Refactoring: Improving the Design of Existing Code" [12]. In chapter 3, they identified 22 code smells as indicators of possible design flaws that may lead to maintainability issues.

In this section we present just a few of the more important smells that we use extensively in this paper. We refer the reader to the Appendix at the end of paper (page 10, past the Bibliography) for a full description of the 22 code smells analyzed in this paper.

Data Class: The Data Class refers to a class with an interface that exposes data elements instead of providing any substantial functionality. This data is usually manipulated by other classes in the system. This means that data and behavior are not in the same scope, which indicates a poor data-functionality proximity. By allowing other classes to access its internal data, Data Classes contribute to a design that is fragile and hard to maintain [12].

Data Clumps: Data Clumps are large groups of parameters that appear together in the signature of many operations. This is a sign that they form a concept which is not yet explicitly represented, which hampers understanding of operations [12].

Blob Class: The Blob Class is an excessively large and complex class, which contains at least one Blob Operation. Such a class is also less cohesive and too strongly coupled to other classes in the system. This makes it very hard to understand and maintain [12].

Code Duplication (internal/external): Code Duplication refers to groups of operations which contain identical or slightly adapted code fragments. By breaking the essential *Don't Repeat Yourself* (DRY) rule, duplicate code increases maintenance effort, including the management of changes and bug-fixes. Moreover, the code base gets bloated. Based on different refactoring approaches we distinguish two cases: (i) internal duplication involving methods that belong to to the same scope (class or module) and (ii) external duplication that refers to unrelated operations [12].

Feature Envy: Feature Envy refers to an operation that is excessively manipulating data external to its definition scope. In object-oriented code this is a method that uses many data members from a few other classes instead of using the data members in its definition class [12].

God Class: The God Class is an excessively complex class which gathers too much and non-cohesive functionality and heavily manipulates data members from other classes [12].

3. LITERATURE SURVEY ON CODE SMELLS

To get an overview of the current state of the art research on code smells in Android application, we conducted a literature survey. We targeted all full conference papers related to code smells from 2008 (the year Android was launched) to 2015 (inclusive) that appeared in top conferences in Software Engineering: ICSE, FSE, OOPSLA/SPLASH, ASE, ICSM/ICSME, MSR, and ESEM.

Starting from the proceedings, we searched for a set of keywords including code smell, smell, bad smell, design flaw, bad pattern etc. From the resulting 52 papers on code smells, we analyzed how many and what kinds of code smells they studied, and which types of projects (Android / desktop) they used in their study.

Table 1 summarizes our findings from the literature survey of code smells papers. Notice that most papers (47) target code smells in desktop applications and only 5 include Android projects in their corpus (no papers analyzed other mobile frameworks, e.g., iOS or Windows Phone).

Table 1. Summary of code smells literature (2008 to 2015).

Conference	# Papers on code smells	# Android Papers	# Java Desktop Papers
ICSE	17	1	16
FSE	1	0	1
OOPSLA/ SPLASH	1	0	1
ASE	6	3	3
ICSM/ ICSME	19	0	19
MSR	3	1	2
ESEM	5	0	5
Total	52	5	47

Moreover, the size of the corpus used in the previous work on Android code smells varies from 5 to 100 applications per paper, with an average size of 46 applications.

A second surprising finding was how few kinds of code smells are investigated in each paper in the literature. For the Android papers the average number of kinds of code smells investigated in a paper is 5 and for the desktop papers it is 6.5. However, for several years, both the research community and the practitioners had access to extensive catalogs of code smells. The ones included in Martin Fowler's book [12] have been around for almost 20 years, and those in Lanza and Marinescu's book [23] for 10 years. Moreover, mature tools that can detect more than 20 kinds of code smells have been around for more than 10 years [25].

Puzzled by these results, we set out to discover what kinds of code smells have attracted the attention of the research community. Table 2 ranks the popularity of Android code smells addressed in the literature. The first column presents the code smell name, the second presents the ranking (based on how many papers address this smell). The third presents a ranking of code smells based on the frequency of the smell in our corpus of 500 Android applications.

Surprisingly, Table 2 shows there is a big gap between the studied code smells in the literature and the code smells in real Android applications. As the data shows, some code smells, such as Blob Class, have received lots of attention in the literature, but many common code smells have received little or no attention. For example, in our corpus of 500 Android apps, we found that Data Clumps and Cyclic Dependencies appear as the second and third most frequent code smells. Despite the wide prevalence of these code smells in Android applications, the previous work on Android code smells never investigated them.

A similar gap between literature and practice was also confirmed by Ahmed et al. [2], in an analysis of the popularity of code smells in desktop applications (See Table 3).

Table 2. Comparison of rankings based on the number of research papers dealing with Android code smells, and our analysis of code smells in 500 Android apps.

Code Smell	Literature Ranking	Our Android Application Ranking
Blob Class	1	16
Feature Envy	2	4
Long Method	3	21
Shotgun Surgery	4	17
Parallel Inheritance	4	22
Divergent Change	4	23
Internal Duplication	5	19
Data Class	6	1
God Class	6	8

Table 3. Comparison of rankings based on the number of research papers dealing with Java desktop code smells, and those in our large corpus [1] of real-world applications.

Smell	From Literature	From Projects
Duplication	1	5
Feature Envy	2	6
Refused Parent Bequest	3	15
Data Class	4	2
Blob Operation	5	4
Blob Class	5	11
Shotgun Surgery	6	17
Data Clumps	7	1
SAP Breakers	8	7
Intensive Coupling	8	9
Schizophrenic Class	8	10
Unstable Dependencies	8	12
Tradition Breaker	8	13
Message Chains	8	16
Distorted Hierarchy	8	18
Unnecessary Coupling	8	19
God Class	9	8
Cyclic Dependencies	10	3

A possible explanation for the big discrepancy between Android code smells in literature and those in real-world Android applications (refer to Table 2) is because the researchers selected those code smells (second column Table 2) that were previously well investigated in the desktop literature (see second column of Table 3).

4. METHODOLOGY

Subsection 4.1 presents our corpus and selection criteria. Subsection 4.2 presents the criteria used for selecting a code smell detection tool to apply to our corpus. Subsection 4.3 presents our Data collection process and 4.4 represents our analysis procedure.

4.1 Project Selection Criteria

This section describes the corpus that we used for our study on code smells.

We first collected a corpus of Android projects. We chose Android over other mobile platforms due to Android being, at the writing of this paper, the most popular operating system for smartphones[5] . As of July 2015, Google Play Store has more than 1.6 million applications[6]. Another reason is the availability of a large variety of Android projects with source code hosted in open source repositories. Given these reasons, we think our corpus is representative for the domain of mobile applications.

For comparison, we also gathered a corpus of desktop Java projects. We settled on Java because (i) it is the most popular programming language[7], (ii) using Java allows for a valid comparison with Android projects that are mostly written in Java, and (iii) there are more code smell detectors for Java than other languages.

To eliminate potential sampling bias [27] and to ensure a diverse sample, we populated our corpus with random projects from GitHub. For the Android corpus, we searched GitHub using "android" as a keyword and manually verified that the project was an Android project.

We started with a corpus of 750 Android projects and 1,000 desktop projects. We then filtered out small projects (those that have fewer than 15 files, or fewer than 500 LOC). This filtering was essential to ensure that the projects we analyzed are representative for real-world projects, and are not throw-away code. After filtering, our final data set contained 500 Android projects with 6.7M LOC, and 750 desktop projects comprising 16M LOC.

We manually grouped our Android projects into 6 categories, identified by other researchers [2, 6], reflecting the categories used by the Google Play Store. The categories are: Development, Audio & Video, Communication, Home & Education, Security & Utilities, and Other. The Development category includes projects such as APIs, libraries, or widgets used in developing applications. The Other category includes all the remaining applications that do not fall into the previous categories and includes games, system administration and business/enterprise apps. We combined the previously mentioned application sub-categories into the Other category because the number of projects in each sub-category in our corpus is much smaller than the top categories (Security & Utilities, Development, etc.). Figure 1 shows the distribution of the Android applications in our corpus.

We were expecting that Games will make up a top-level category. However, the presence of few games in our randomly selected sample of 500 Android apps could indicate that open-source games are as not prevalent in GitHub when compared to the Google Play Store. Since games are one of the most profitable category of mobile apps, they are featured widely in the closed-source ecosystem.

4.2 Tool selection

To perform our code smell analysis across Android and desktop applications, we had to select which code smell detectors to apply. We decided to use inFusion [19], a commercial tool. We favor inFusion for several reasons. First, inFusion detects 22 different code smells (see Table 3 in Appendix). Second, a previous study by Ahmed et al. [2] showed that inFusion has a very high precision and recall (84% and 100% respectively). Third, inFusion scales to analyzing large projects. Moreover, there are many studies [10,11, 17] that use inFusion as their code smell detection tool. We considered using other code smell detection tools, but we chose inFusion for the reasons cited above. We ran inFusion on our Android and desktop corpora.

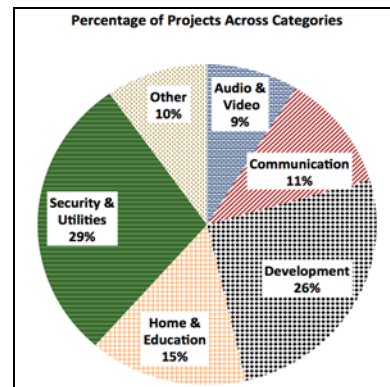

Figure 1. Distribution of Android application categories in our corpus.

4.3 Data collection

To analyze code smells in Android applications, we could either run code smell detectors on the source code or on the decompiled .apk files. However, it is possible that the compiler optimizes away some of the code smells (for example, due to dead code elimination). To verify our assumption, we collected 10 Android projects from GitHub and built .apk files. Then we decompile them and run inFusion on both versions: the original source code and the decompiled code. The results confirmed our assumption.

We found important differences between the code smells reported by inFusion in the two setups. First, the number of code smells reported is different, as can be seen in Table 4. While in most cases, the number of code smells in decompiled apk files was smaller than in the original source code, in one case the situation was reversed. Furthermore, we noticed that the kinds of code smells reported by inFusion in the two setups were different. Table 4 shows an example of one application (Cube app) where the number and the kinds of code smells were different. After running inFusion on the original source code of Cube app, it detected 2 occurrences of SAP

[5] http://www.gartner.com/newsroom/id/3061917

[6] http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[7] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Breakers (see definition in Appendix) and 2 occurrences of Cyclic Dependencies. However, when running InFusion on the decompiled code, it found surprisingly different code smells: 3 occurrences of Data Clumps. One explanation could be that the compiler may optimize away some code smells. Thus, we decided to only run the code smell detector on the original Android application source code. This allows a fair, more direct comparison with the desktop applications.

Table 4. Code smells detected in Android Cube app

Original source code	Decompiled code from .apk file
SAP Breakers: 2 occurrences - one involving 10 files - one involving 7 files Cyclic Dependencies: 2 occurrences	Data Clumps: 3 occurrences

For each application in our corpora, we ran inFusion and recorded the number and type of code smells. Moreover, to shed light on any potential differences, we collect these **additional metrics**:

- Application size in LOC,
- Number of files,
- Number of developers,
- Number of commits,
- Number of bug fix commits,
- Number of new feature commits,
- Number of merges,
- Number of core committers,
- The age of the projects.

We tracked these metrics, as Ahmed et al. [1] showed that file count, number of core developers, number of commits, project age, and number of lines deleted correlate with the number of code smells in Java applications. Tufano et al. [40] also found that most code smells are introduced during bug fixes or new feature addition.

In order to distinguish between commits that were bug fixes and those that are new features, we classified 457,510 Android commits and 355,500 desktop commits using a multinomial Naïve Bayes classifier [29] with Laplace smoothing. We used tf-idf as the feature set of the classifier. This classifier is representative of the best-in-class for this problem domain, and several other researchers used it [3, 16]. We used 1500 commits as training data (750 BF and 750 NF) both from Android and desktop projects and used 10 fold cross validation. Precision and recall of our classifier was 0.70 and 0.71 respectively.

4.4 Data Analysis

To answer our first research question we collected the total number of code smells after each commit. We normalized the smell count using feature scaling (1), which gives us a score between 0 and 1.

$$\text{Rescaled value} = \frac{x - \min(x)}{\max(x) - \min(x)} \quad (1)$$

Where: x = each data point
min(x) = The minima among all the data points in one column.
max(x) = The maxima among all the data points in one column.

Previous studies have shown that normalizing the smell count using the project size reduces the bias of larger projects on the overall smell count [39]. For our study this was not necessary. Our aim was to identify general trends across projects, not to look at differences between them. There was therefore no need to normalize based on project size.

5. RESULTS

In this section, we present the results for each research question.

5.1 RQ1: What is the variety, density, and distribution of code smell in Android application? How does it compare with desktop applications in terms of code smells?

Variety. We found that both the Android and the desktop corpus contain instances of all 20 code smells (out of 22 kinds of code smells detected by inFusion). That was a surprising finding given that Android applications have different structure and workflow (event-driven). Possible explanations for this similarity include (i) that both populations are written in Java and make use of the same

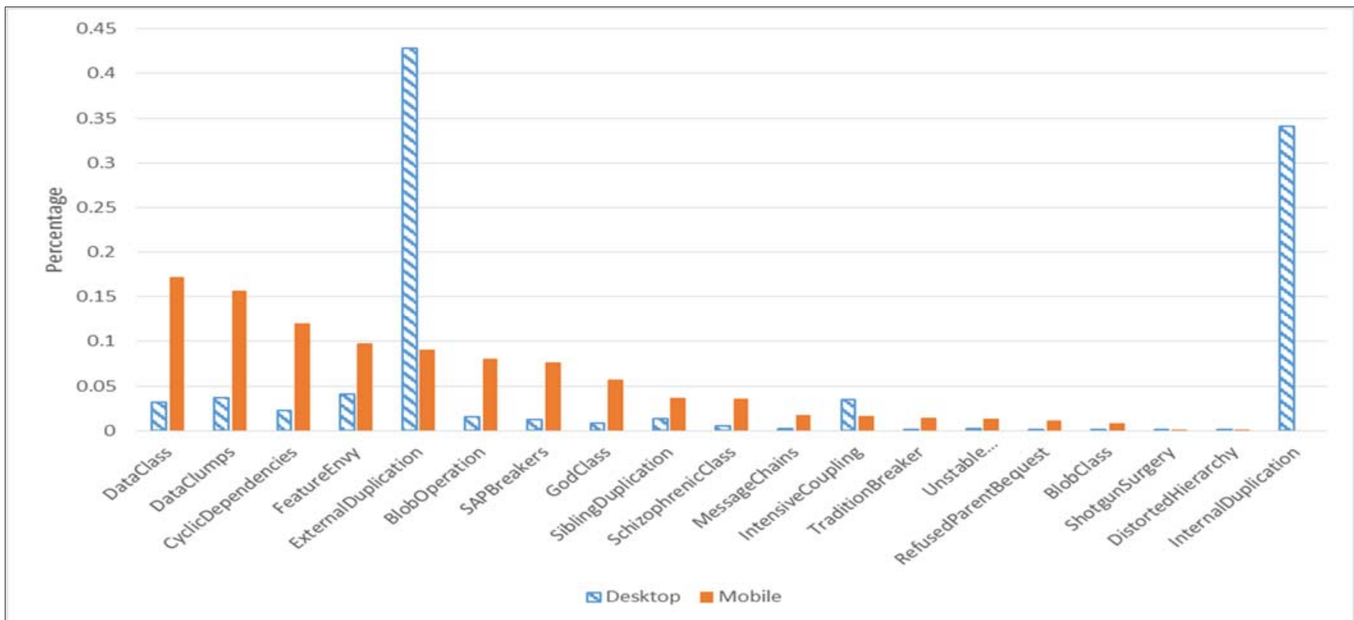

Figure 2. Distribution of code smells across Android and desktop applications.

OO features, (ii) many Android programmers have a background of developing desktop Java applications, so they are prone to making the same mistakes and tradeoffs.

Density. Next we analyzed the density of code smells in mobile and desktop applications. We compare the numbers of code smells per thousand lines code (KLOC) in mobile and desktop applications. The average number of code smell per KLOC in mobile applications is 173.39 and in desktop application 190.41. However, this difference was not statistically significant (Welch Two Sample t-test, t = -0.2201, df = 32.593, p-value = 0.8272). We used Welch two sample t-test, because the number of projects in Android and Desktop applications are not equal.

Distribution. We found important differences in the distribution of code smells between Android and desktop application. We show these differences in Figure 2, where Android code smells are shown with solid orange bars and desktop code smells blue patterned bars. We performed a statistical significance test (using a Welch two-sample t-test – since the size of our Android and desktop corpus is different). We found that the difference in the distribution of code smells is statistically significant (Welch Two Sample t-test,$p < 0.05$) for all but two code smells (Distorted Hierarchy and Message Chains).

As Figure 2 shows, the most common code smells in Android are Data Class and Data Clumps. In desktop applications, the main culprits are External Duplication and Internal Duplication. One explanation could be that most of the desktop applications have been around for a longer period than mobile applications. Indeed, when studying the age of the projects in our corpus, we found that the average age of desktop applications is 3.5 years, whereas the average age of mobile applications is 2 years. Therefore, according to the well-known laws of software evolution, the quality might decay as projects get older and as contributors copy code. A possible explanation for the excessive number of Data Classes and Data Clumps in Android is that developers declare several variables that they think will be useful later, in future releases, but they end up not using them. In Android there is significantly less duplication than in desktop applications. A possible explanation is that developers aim to make their applications more lightweight to reduce the memory and the code footprint, since mobile devices have much fewer resources than desktop computers.

To shed further light into why the distribution of code smells is different between Android and desktop applications, we looked for correlations with other metrics (those introduced in Section 3.4), such as code/team size, number of commits that contain bug fixes or new features.

We performed a linear regression analysis to find out which factors have an effect on the total number of code smells for both android and non-android. We did linear regression analysis with the intercept $\beta 0$ set to zero (absence of files or other variables results in zero code violations). For android, according to the final model we got from the analysis, total number of smells in a project is related to number of line, number of files, total bug fix commits, total new feature commits, merge counts and number of core developers. Our model discarded number of developers, total commits and duration of the project as significant factor. Table 5 contains the details of linear regression analysis of android. The linear regression model that we found have R-squared value of 0.5342 meaning that a handful of project factors can account for more than 50% of the variance in the sample.

On the other hand, according to the final model for non-android total number of smell per project is related to total number of lines, total number of developers, merge counts, total number of core developers and duration. For non-android our model discarded number of files, total commits, total bug fix commit and total new feature commits of the project as significant factor. Table 6 contains the details of linear regression analysis of non-android. The linear regression model that we found have R-squared value of 0.1033 meaning that a handful of project factors can account for more than 10% of the variance in the sample.

Table 5. Linear regression model for Android

Title	Estimate	Std.Error	t-value
# of Lines	0.65098	0.05883	11.066
# Bug fix commits	0.51968	0.22833	2.276
# New feature commits	-0.52182	0.24658	-2.116
# of Files	0.18626	0.06569	2.836
Number of Merges	-0.23315	0.11426	-2.041
Number of Core Devs	0.10615	0.02677	3.965

Table 6. Linear regression model for desktop applications

Title	Estimate	Std.Error	t value
Number of Lines	0.1068	0.0574	1.861
Number of developers	-0.18849	0.0943	-1.999
Merge counts	-0.11817	0.07589	-1.557
Total core developers	0.39871	0.07736	5.154
Duration	0.08624	0.04944	1.744

5.2 RQ2: What is the distribution of code smells across categories of mobile apps?

To answer this question, we categorized all our Android applications into one of six categories: Audio & Video, Communication, Development, Home & Education, Security and Utilities, and Other.

Figure 3 shows the code smells are distribution among different Android application categories. Similar to Figure 2, we found that the Data Class is the most common code smell across all 6 categories. We can also see that Data Class is the highest in Home & Education category, and Data Clump is the highest in Development category. Moreover, we conducted a fine-grained analysis between the distributions of code smells across the 6 categories. Table 7 shows that we found statistically significant differences between certain categories.

Table 7. Difference in code smell distribution among Android categories.

Categories	P-value
Audio & video AND Others	0.02128
Communication AND Home & education	0.009854
Communication AND Others	0.002738
Development AND Home & education	0.0489
Development AND Other	0.01006
Home & education AND Security & utilities	0.0111
Security & utilities AND Other	0.003959

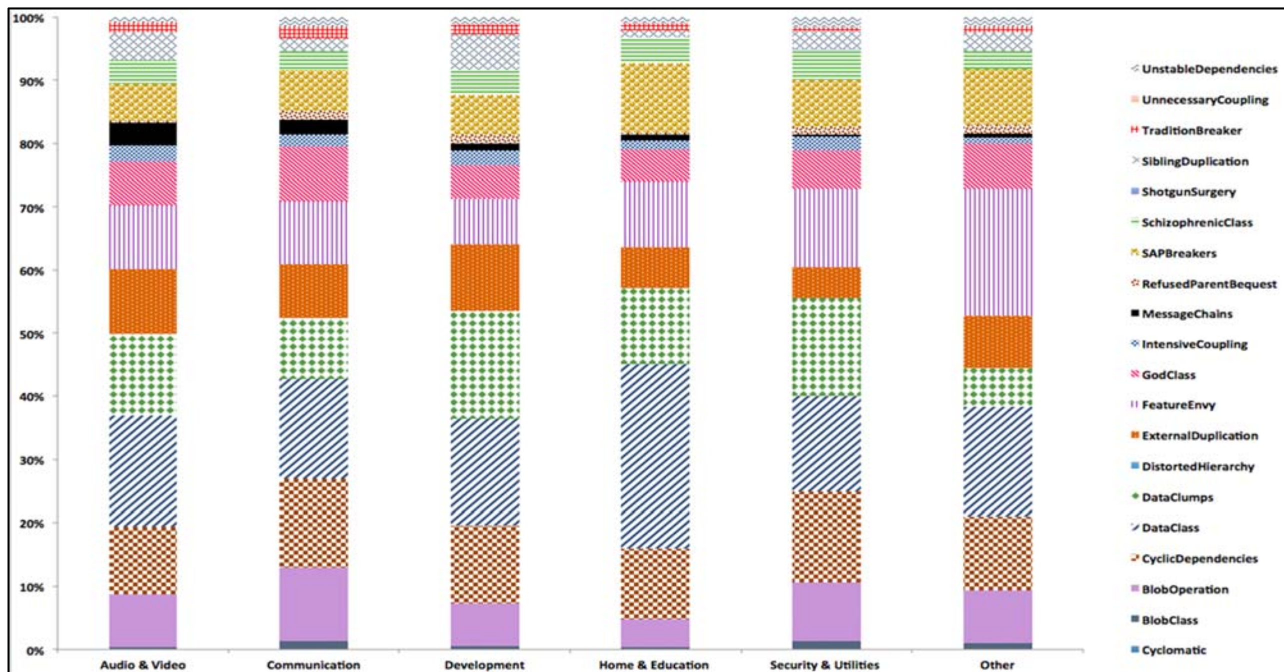

Figure 3. Distribution of code smells across different categories of Android applications

6. THREATS TO VALIDITY

Construct: *Are we asking the right questions?* Our goal was to determine whether the development styles of mobile application and desktop application developers would result in differences in code smells. Since our questions compare the two populations in regard to variety, density, and distribution of code smells, these measures help us achieve our goal.

Internal: *Is there something inherent to how we collect and analyze code smells that could skew the accuracy of our results?* Since we are relying on inFusion to detect smells, the accuracy of our results depends on the accuracy of this tool. InFusion uses threshold-based detection strategies, the efficacy of which has been evaluated in a previous study [2]. However, the tool has not been evaluated in all contexts. InFusion uses static program analysis to identify smells, and research [31] shows that code smells that are "intrinsically historical" such as Divergent Change, Shotgun Surgery, and Parallel Inheritance are difficult to detect solely through static analysis. The number of instances such smells might be different when historical information is taken into account. Nevertheless, inFusion is a commercial-quality tool, and to date detects the highest variety of code smells.

Moreover, there are large differences in the number and size of the projects that we used in our Android and desktop corpora. These differences might make it harder to compare the two populations. To account for such differences, we normalized the number of code smells using the standard feature scaling [2] used in the literature, and also by the size of projects.

External: *Are the results generalizable to proprietary software development, to other programming languages, or to other mobile platforms?* We use GitHub to assemble a corpus of 500 Android and 750 desktop applications. These span a wide range of domains, from tools, games, image and audio processing, web systems, social media etc., to third party libraries. They are developed by different teams with 7,621 contributors from a large and varied community. We believe that (i) the relatively large and varied corpus and (ii) the fact that we did not target the best or the worst applications might make the results applicable to proprietary development as well. However commercial applications that are maintained by professional teams and subject to organizations guidelines might display a different distribution of code smells.

All our subject programs are written in Java, but code smells are not specific to Java. Any programs written in programming language that provide objects as first class citizens will be prone to the same code smells (though some code smells are specific to object oriented languages).

With respect to other mobile platforms, we analyzed applications from Android, which has the largest market share. The iOS ecosystem contains mobile applications written in Objective C, whereas the Windows Phone applications are written in C#. Given that these languages are all object-oriented, we expect they would exhibit the same code smells.

Reliability: *Are our results replicable?* We collected our data from applications available on GitHub. The list of the applications that we used, and the specific version is available on our website[8] Furthermore, inFusion is a commercially available tool. We presented our experimental setup in the methodology (Section 3), making replication of our study possible.

7. IMPLICATIONS

We present practical implications of our study for researchers, tool builders, and application developers.

7.1 Researchers

As Figure 2 shows, the desktop applications are dominated by just a couple of code smells (External Duplication and Internal Duplication). In desktop applications, more than 75% of instances

[8] http://web.engr.oregonstate.edu/~mannanu/AndroidProjects.txt

of code smells will be related to duplication. In contrast, Android applications exhibit a large variety of code smells: despite Data Class and Data Clumps being the pack leaders, the other code smells are very well represented. Thus, a researcher studying code smells has a better chance of finding instances of many other kinds of code smells in Android applications.

The large variety of code smells in Android applications is also good news for educators. In Software Engineering education, the most effective illustrations of SE concepts and abstractions are those that incorporate real-world examples and artifacts. Students learn and educators teach SE design principles through both positive and negative examples. Robillard and DeLine [38] study what makes large APIs hard to learn and conclude that one of the important factors is the lack of usage examples.

Educators can illustrate many design principles by showing both well-designed programs and those that exhibit many code smells. Using Android applications as subject case studies is guaranteed to provide a variety of code smells. Moreover, students might also prefer examples from the mobile domain given, which rise and allure of mobile programming.

Many Android applications do not publish their source code, only the binary apk files. However, many tools can decompile apk files into source code. Thus, it is tempting for researchers of code smells to use Android apks as input to their analysis. In our formative study we found that the compilation/decompilation process can dramatically affect the number and the kind of code smells displayed. We encourage researchers to only use the source code as input when analyzing code smells.

Our literature survey (Section 3.1) and the data that we found in a large corpus of Android and desktop applications (Section 4) shows a large discrepancy between the code smells that received attention in the literature and those that dominate in real-world applications. Given the wide variety of code smells in Android applications, we encourage researchers to study many more kinds of code smells. If resources are limited and researchers need to prioritize which code smells to study, we recommend that they narrow the selection based on prevalence of code smells in real-world applications, or their impact on said applications.

7.2 Tool builders

The good news for code smell tool builders is that code smell detectors designed for desktop software can be used for mobile applications. However, given the wider variety of code smells in Android applications, we encourage tool builders to allocate support for robust implementation of several detection strategies.

Tool builders as well as researchers in the refactoring community can use our findings to target Android-specific tools. To date, very few refactorings are data-related, but we have seen that Data Class and Data Clumps appear frequently in Android applications.

Moreover, there is a gap between tools that detect code smells, and tools that eliminate code smells (e.g., refactoring tools). We think the high presence of code smells points to the fact that developers might not act upon the warnings given by code smell detection tools, unless they have either (i) a high return on investment (e.g., eliminating the smell has an immediate value), or (ii) tools make it easier to eliminate the smell. While developers might not see the immediate benefits of eliminating code smells (as smells usually have long-term effects upon code maintenance), we encourage tool builders to close the gap between detection and correction of code smells.

7.3 Application developers

The data in Figure 2 shows that Android applications display a larger variety of code smells than desktop applications. Developers should educate themselves about the kinds of code smells that occur in mobile applications, and how to mitigate them. Or even better, being conscious about code smells when editing code might help avoid them altogether.

Given that the code smell detectors used in desktop Java applications are effective for Android applications, developers don’t need to learn new tools.

From our findings, we can see that mobile app developers are more prone to introduce Data Classes and Data Clumps into their code. We warn developers that they need not prematurely introduce variables they think they might use in later versions, but instead embrace a more Agile approach of adding a class variable when it is needed.

8. RELATED WORK

We group the related work into the following areas: (i) studies that analyze code smells in desktop applications, (ii) code smells that are specifically defined for Android applications, (iii) studies that use Android applications in their corpus.

8.1 Code smells in desktop applications

Researchers have identified a variety of techniques for identifying design degradation using static analysis. Some researchers [8, 9, 30] assess degradation by analyzing a single, static version of the software. Other researchers [21, 22, 31] designed techniques that rely on the evaluation of successive versions of a software systems.

Ahmed et al. [2] looked at 220 open source projects and analyzed the presence and evolution of code smells. Their results confirmed that ignoring code smells leads to “software decay”. Moreover, this study pointed out that some code smells that appear frequently in their examined applications received less attention by the research community.

8.2 Android-specific code smells

Hecht et al. [13,14] designed a code smell detection tool named PAPRIKA to detect 8 code smells by analyzing the bytecode of Android applications. 4 of these 8 code smells were Android-specific: Internal Getter/Setter, Member Ignoring Method, No Low Memory Resolver, and Leaking Inner Class. They found that the Android-specific code smells occur more often than the others. However, their validation of PAPRIKA was based on only 15 applications. In follow-up work, Hecht et al. [15] updated the tool to detect smells from different versions of the App and calculate an evolutionary “quality score” based on the presence of code smells. The updated PAPRIKA detects 7 code smells instead of 8. One Object Oriented (OO) code smell, Swiss Army Knife, was removed, probably due to infrequent detection in applications [13,14]. Their results show different trends for “quality evolution” but none can be generalized. Although Hecht et al. [13,14,15] compared the existence of OO vs. Android code smells, only 4 code smells of each category were investigated. In regards to our project, we don’t include the detection of Android specific smells, we do include all 22 OO code smells.

Reimann et al. [35] presented a catalog listing 30 Android specific code smells, along with a refactoring tool called Refactory. The catalog was assembled to form a formal definition for Android code smells. The authors categorized these smells based on what qualities smells affect and the context they occur in. The tool

Refactory, as described by the authors, employs a role-based approach to support context-specific refactoring. Refactory was implemented to the code smells covered in their catalog, and then generate "fixes," suggestion to the developer on how to get rid of the code smells detected.

8.3 Studies that use Android applications

Verloop et al. [41] validated four OO code smells detection tools on Android applications; these were JDeodorant, Checkstyle, PMD and UCDetector. The results show interesting code smell ratios for Android application code (i.e. code that inherits from the android framework), when compared to smells found in non-Android application code. Nonetheless, the author only considered 14 applications in his study and 6 OO code smells.

Tufano et al. [40] looked at when code smells appear in the code and the circumstances associated with their appearance. They tracked the evolution of 200 open source project (70 of which were Android apps). Results showed that if a file was smelly, the smell was most likely introduced when the file was first created. As to the circumstances, Tufano et al. found that when approaching a deadline, developers are more likely to add a smell when adding a new feature or enhancing an existing one.

Delchev and Harun [7] were interested in how frequently code smells are encountered and the severity of their effect. They conducted a survey where they asked 73 developers about 10 code smells. They asked how frequently the developer encountered a smell and how likely they were to refactor such smells? The ten smells were: Data Class, Long Parameter List, Switch Statements, Message Chains, Primitive Obsession, Data Clumps, Refused Bequest, Feature Envy Shotgun Surgery and Long Method. Authors grouped the results based on project domain, project language and developer experience. With regards to Android projects, the survey found that developers faced Long Methods smells more than other smells, but Shotgun Surgery was more likely to be refactored. Also, frequency and severity varied relative to programming language. As for developer experience, they found that the more experienced the developer, the less likely they were to face smells. However, when these more experienced developers did, they had a higher tendency to refactor that smell.

9. CONCLUSIONS

This empirical study compares code smells in Android vs. desktop applications in terms of their variety, density, and distribution. We used an open-source corpus consisting of 500 Android and 750 desktop applications. Although we expect to find important differences in the kinds and density of code smells, our results show that Android and desktop applications are very similar in terms of the code smells that are detected by InFusion.

However, we found that the distribution of code smells varies significantly. Whereas in desktop applications the code smells are dominated by two smells (External Duplication and Internal Duplication), Android applications display a more diverse set of code smells.

Our study has practical value. For researchers we present several pitfalls they can avoid when studying code smells. For tool builders we present new areas of development. For application developers, we make them aware of the large variety of code smells that can easily creep into Android applications.

Our literature survey also shed light on the gap between the code smells studied in the literature and the code smells that appear in practice. Researchers can easily become biased or develop blind spots. We hope that our study will also help others to concentrate on the right code smells.

We hope that this research encourages the community to further investigate the important domain of mobile applications and how they are different from traditional desktop software.

10. REFERENCES

[1] Ahmed, I., Ghorashi, S., & Jensen, C. (2014). An Exploration of Code Quality in FOSS Projects. In Open Source Software: Mobile Open Source Technologies (pp. 181-190). *Springer Berlin Heidelberg*.

[2] Ahmed, I., Mannan, U. A., Gopinath, R., & Jensen, C. "An Empirical Study of Design Degradation: How Software Projects Get Worse Over Time*." Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on. IEEE, 2015*.

[3] Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., & Guéhéneuc, Y. G. (2008, October). Is it a bug or an enhancement?: a text-based approach to classify change requests. *In Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds* (p. 23). ACM.

[4] Brown, W. H., Malveau, R. C., & Mowbray, T. J. (1998). AntiPatterns: refactoring software, architectures, and projects in crisis.

[5] Cunningham, W. (1992, December). The WyCash portfolio management system. *In ACM SIGPLAN OOPS Messenger*, Vol. 4, No. 2, (pp. 29-30).

[6] De Souza, L. B. L., & Maia, M. D. A. (2013, May). Do software categories impact coupling metrics*?. In Proceedings of the 10th Working Conference on Mining Software Repositories* (pp. 217-220). IEEE Press.

[7] Delchev, M., Harun, M. F. (2015). Investigation of Code Smells in Different Software Domains. *Full-scale Software Engineering, 31*.

[8] Deligiannis, I.; Shepperd, M.; Roumeliotis, M.; Stamelos, I. (2004). "An empirical investigation of an object-oriented design heuristic for maintainability*". In The Journal of Systems and Software 72 (2), (pp.129-143).*

[9] Deligiannis, I.; Stamelos, I.; Angelis, L.; Roumeliotis, M.; Shepperd, M. (2003, February). "A controlled experiment investigation of an object oriented design heuristic for maintainability" *In Journal of Systems and Software, Vol.65, No .2, (pp.127-139)*.

[10] Ferme, V., Marino, A., & Fontana, F. A. (2013). Is it a Real Code Smell to be Removed or not?. In International Workshop on Refactoring & Testing (RefTest), co-located event with XP 2013 Conference. *(RefTest), co-located event with XP 2013 Conference. 2013.*

[11] Fontana, F. A., & Zanoni, M. (2011, March). On investigating code smells correlations. In Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on (pp. 474-475). IEEE.

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

[13] Hecht, G. An Approach to Detect Android Antipatterns. *In ICSE 2015-ACM SRC*.

[14] Hecht, G., Duchien, L., Naouel, M., & Rouvoy, R. Detection of Anti-patterns in Mobile Applications. In COMPARCH 2014.

[15] Hecht, G., Omar, B., Rouvoy, R., Moha, N., & Duchien, L. (2015, November). Tracking the Software Quality of Android Applications along their Evolution. *In 30th IEEE/ACM International Conference on Automated Software Engineering* (p. 12). IEEE.

[16] Herzig, K., Just, S., & Zeller, A. (2013, May). It's not a bug, it's a feature: how misclassification impacts bug prediction. *In Proceedings of the 2013 International Conference on Software Engineering* (pp. 392-401). IEEE Press.

[17] *Hozano, M., Ferreira, H., Silva, I., Fonseca, B., & Costa, E. (2015, April). Using developers' feedback to improve code smell detection. In Proceedings of the 30th Annual ACM Symposium on Applied Computing (pp. 1661-1663). ACM.*

[18] Hunt, A., & Thomas, D. (2000). The pragmatic programmer: from journeyman to master. Addison-Wesley Professional.

[19] InFusion. Retrieved from URL: http://www.intooitus.com/inFusion.html.

[20] Izurieta, C., & Bieman, J. M. (2008, April). Testing consequences of grime buildup in object oriented design patterns. *In Software Testing,Verification, and Validation, 2008 1st International Conference on (pp. 171-179). IEEE.*

[21] Kagdi, H., Collard, M. L., & Maletic, J. I. (2007). "A survey and taxonomy of approaches for mining software repositories in the context of software evolution". *In Journal of Software Maintenance and Evolution: Research and Practice, Vol.19, No. 2, (pp. 77-131)*.

[22] Kagdi, H., Gethers, M., Poshyvanyk, D., & Collard, M. L. (2010, October). "Blending conceptual and evolutionary couplings to support change impact analysis in source code". *In 17th Working Conference on Reverse Engineering, (pp. 119-128).*

[23] Lanza, M., & Marinescu, R. (2007). "Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems". *Springer Science & Business Media.*

[24] Li, W., & Shatnawi, R. (2007)." An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution". *In Journal of Systems and Software, Vol.80, No.7, (pp.1120-1128).*

[25] Marinescu, C., Marinescu, R., Mihancea, P. F., & Wettel, R. (2005). iPlasma: An integrated platform for quality assessment of object-oriented design. In ICSM (Industrial and Tool Volume).

[26] Martin, R. C. (2003). Agile software development: principles, patterns, and practices. Prentice Hall PTR.

[27] Martin, W., Harman, M., Jia, Y., Sarro, F., & Zhang, Y. (2015, May). The app sampling problem for app store mining. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on* (pp. 123-133). IEEE.

[28] McIlroy, S., Ali, N., & Hassan, A. E. (2015). Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. *Empirical Software Engineering, 1-25*.

[29] Murphy, K. P. Naive bayes classifiers. *University of British Columbia* (2006).

[30] Olbrich, S., Cruzes, D. S., Basili, V., & Zazworka, N. (2009, October). 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 (pp. 390-400)*

[31] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., & Poshyvanyk, D. (2013, November). Detecting bad smells in source code using change history information. *In Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on (pp. 268-278). IEEE*.

[32] Palomba, F., Di Nucci, D., Tufano, M., Bavota, G., Oliveto, R., Poshyvanyk, D., & De Lucia, A. Landfill: an Open Dataset of Code Smells with Public Evaluation.

[33] Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; Poshyvanyk, D.; De Lucia, A., Mining Version Histories for Detecting Code Smells. In *IEEE Transactions on Software Engineering*, vol.41, no.5, pp.462-489, May 1 2015.

[34] Reimann, J., & Aβmann, U. (2013, December). Quality-Aware Refactoring For Early Detection And Resolution Of Energy Deficiencies. *In Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing (pp. 321-326). IEEE Computer Society*.

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

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

[37] Riel, A. J. (1996). Object-oriented design heuristics (Vol. 338). Reading: Addison-Wesley.

[38] Robillard, Martin P., and Robert Deline. A field study of API learning obstacles. *Empirical Software Engineering* 16.6 (2011): 703-732.

[39] Schumacher, J., Zazworka, N., Shull, F., Seaman, C., & Shaw, M. (2010, September). Building empirical support for automated code smell detection. *In Proceedings of the International Symposium on Empirical Software Engineering and Measurement* (p. 8). ACM.

[40] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., & Poshyvanyk, D. (2015). When and Why Your Code Starts to Smell Bad. *ICSE 2015*.

[41] Verloop, D. (2013). Code Smells in the Mobile Applications Domain (Doctoral dissertation, TU Delft, Delft University of Technology).

[42] Yamashita, A., & Moonen, L. (2013, October). "Do developers care about code smells? An exploratory survey". *In Proceedings of the 20th Working Conference on Reverse Engineering (WCRE), (pp. 242-251.*

Appendix

Table 3. List of code smells identified by inFusion.

Smells	Definition
Cyclic Dependencies	Cyclic Dependencies are violations of the Acyclic Dependencies Principle formulated by Robert Martin [26] as "The dependency structure between packages must be a Directed Acyclic Graph (DAG). That is, there must be no cycles in the dependency structure". The design flaw applies to dependencies between subsystems of a system. If two or more subsystems are involved in a cycle, maintaining or reusing any one of those subsystems in isolation will be harder or impossible [19,24].
Brain Method	Brain Methods tend to centralize the functionality of a class, in the same way as a God Class centralizes the functionality of an entire subsystem, or sometimes even a whole system.
Data Class	Data Classes are "dumb" data holders, without complex functionality, but which are usually heavily relied upon by other classes in the system. Data classes are the manifestation of a lacking encapsulation of data, and of a poor data-functionality proximity. By allowing other modules or classes to access their internal data, data classes contribute to a brittle, and harder to maintain design [12,19,24,37].
Feature Envy	The Feature Envy design flaw refers to functions or methods that seem more interested in the data of other Classes and modules than the data of those in which they reside. These "envious operations" access either directly or via accessor methods. This situation is a strong indication that the affected method was probably misplaced and that it should be moved to the capsule that defines the "envied data" [12,19,24,37].
God Class	The "God class" tends to concentrate functionality from several unrelated classes, while at the same time increasing coupling in the system. The god class itself is probably not very cohesive and because of its size and inherent complexity it will have a clear negative impact on the maintainability of the system [12,19,24,37].
Intensive Coupling	Intensive Coupling is the flaw of an method when a method is tied to many other operations in the system, whereby these provider operations are dispersed only into one or a few classes [12,19,24,37].
Missing Template Method	Two different components have significant similarities, but do not use an interface or a common implementation (the Template Method). [12]
Refused Parent Bequest	Refused Parent Bequest occurs when a derived class uses very few or none of the inheritance-specific members defined by its base class. [12]
Sibling Duplication	Sibling Duplication means duplication between siblings in an inheritance hierarchy. Two or more siblings that define a similar functionality make it much harder to locate errors [4,12,18,19].
Shotgun Surgery	This smell is evident when you must change lots of pieces of code in different places simply to add a new or extended piece of behavior. Whenever a method is called by too many other methods, any change to such a method ripples through the design. Such changes are likely to fail when the number of to-be-changed locations exceeds the capacity of human’s short term memory.[24]
SAPBreakers	Stable Abstraction Breaker is a subsystem (component) for which its stability level is not proportional with its abstractness. This design flaw is inspired by Robert Martin's stable abstractions principle, which states that for well-designed software there should be a specific relationship between two subsystem measures: the abstractness of a subsystem, which shall express the portion of contained abstract types, and its stability, which indicates whether the subsystem is mainly used by other client subsystems (stable) or if it mainly depends on other subsystems (unstable). For short, "a subsystem should be as abstract as it is stable". The problem with subsystems that are heavily used by other subsystems and at the same time are not abstract is that if they change (and they are likely to), potentially all clients must also change. This in turn leads to systems that are hard to maintain. [26 and 19]
Internal Duplication	Internal Duplication means duplication between portions of the same class or module. Thus, the presence of code duplication bloats the class or module and all the clones do not evolve the same way [4,12,18,19].
External Duplication	External Duplication means duplication between unrelated capsules of the system [4,12,18,19].

Smells	Definition
Blob Class	Blob Classes are very large and complex classes, which makes them harder to maintain. Because of their size, they are also more likely to be strongly coupled to other classes in the system and to be non-cohesive [4,12,24,19].
Blob Operation	A Blob Operation is a very large and complex operation, which tends to centralize too much of the functionality of a class or module. Such an operation usually starts normal and grows over time until it gets out of control, becoming hard to read and maintain [4,12,24,19].
Data Clumps	Data Clumps is a design flaw where groups of data that appear together over and over again, as parameters that are passed to operations throughout the system. This represents bad/lacking of encapsulation. Data Clumps are good candidates to become objects. [12]
Message Chains	This smell occur when a long sequence of method calls or temporary variables are required to get some data. Navigating this way means the client is coupled to the structure of the navigation. [4,12,24]
Distorted Hierarchy	A Distorted Hierarchy is an inheritance hierarchy that is unusually narrow and deep. This design flaw is inspired by one of Arthur Riel's [37] heuristics, which says that "in practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short-term memory. A popular value for this depth is six". Having an inheritance hierarchy that is too deep may cause maintainers "to get lost" in the hierarchy making the system in general harder to maintain.[37]
Schizophrenic Class	A "schizophrenic class" is a class that captures two or more key abstractions. It negatively affects the ability to understand and change in isolation the individual abstractions that it captures. [12,37]
Tradition Breaker	This strategy takes its name from the principle that the interface of a class (i.e., the services that it provides to the rest of the system) should increase in an evolutionary fashion. This means that a derived class should not break the inherited "tradition" and provide a large set of services which are unrelated to those provided by its base class. [37,24]
Unstable Dependencies	Unstable Dependencies are violations of Robert Martin's Stable Dependencies Principle (SDP)[26]. The SDP affirms that "the dependencies between subsystems in a design should be in the direction of the stability of the subsystems. A subsystem should only depend upon subsystems that are more or at least as stable as it is". Stability is defined in terms of number of reasons to change and number of reasons not to change for a given subsystem. A subsystem that does not depend on many other subsystems but is depended upon by other subsystems, has few reasons to change and respectively many reasons not to change. [26]