Comparison of Occurrence of Design Smells in Desktop and Mobile Applications

Daniel Ogenrwot
Department of Computer Science
Gulu University
Gulu, Uganda
d.ogenrwot@gu.ac.ug

Joyce Nakatumba-Nabende
Department of Computer Science
Makerere University
Kampala, Uganda
jnakatumba@cis.mak.ac.ug

Michel R.V. Chaudron
Department of Computer Science and Engineering
Chalmers | Gothenburg University
Gothenburg, Sweden
chaudron@chalmers.se

***Abstract*—Design smells are symptoms of poor solutions to recurring design problems in a software system. Those symptoms have a direct negative impact on software quality by making it difficult to comprehend and maintain. In this paper we compare the occurrence of design smells between different technological ecosystems: windows/desktop and android/mobile. This knowledge is significant for various software maintenance activities such as program quality assurance and refactoring. To supplement previous findings, our study aimed at (a) understanding if and how the relationship among design smells differs across windows and mobile applications and (b) determining the groups of design smells that tend to occur frequently together and the magnitude of their occurrence in windows and mobile applications. In this study, we explored the use of statistics and unsupervised learning on a dataset consisting of twelve (12) Java-based open-source projects mined from GitHub. We identified fifteen (15) most frequent design smells across desktop and mobile applications. Additionally, a clustering technique revealed which groups of design smells that often co-occur. Specifically, {*SpeculativeGenerality, SwissArmyKnife*} and {*LongParameterList, ClassDataShouldBePrivate*} are observed to occur frequently together in desktop and mobile applications.**

***Index Terms*—Design Smells, Clustering, Software Quality, Anti-patterns**

I. Introduction

The concept of "design smell" was introduced by Fowler [1] and defined as symptoms of poor solutions to recurring design problems. The symptoms of design smells in a software system are also referred to as *anti-patterns*. Design smells are normally introduced in source code by developers during their daily activities such as the implementation of user requirements, developing important patches or during a "hack" or "workaround" to obtain a sub-optimal solution to existing problems [2]. According to previous studies, the existence of design smells makes programs complex to comprehend, summarize and modify [3], which poses a direct negative effect on software quality [3], [4]. The existence of code smells in any code-base calls for refactoring, which is a technique of restructuring a program without changing its external behavior to ensure that any further development is possible. However, the cost of refactoring becomes expensive in terms of time and resources, especially for today's ever-evolving software platforms.

This work was supported by Mak-Sida Project 381.

The need for effective detection of anti-patterns has attracted a lot of research interests, both from academic and software industry. As such, significant studies towards the realization of effective design smell detection methods and tools have been conducted over the last few years [3], [5], [6]. Software metrics provide the backbone for most anti-pattern detection platforms and approaches. They are applied to evaluate the internal code quality and productivity, as well as maintainability of software. For example; Imran [3] used an unsupervised spectral clustering tool guided by software metrics to detect design smells across 3,306 classes of real-life open-source Java software.

Design smells are detected using a combination of software metrics such as Depth of Inheritance Tree (DIT), Weighted Methods per Class (WMC) and/or Coupling Between Objects (CBO) among others. Understanding the diversity, distribution, magnitude and co-occurrence of various design smells within a source code could provide a good opportunity for optimizing these metrics and consequently improving rule-based design smells detection approaches. Moreover, this knowledge is essential to developers in implementing various design smell control and prevention mechanisms as well as guiding software maintenance activities.

It is upon this motivation that we propose a method based on statistics and machine learning to understand the diversity, distribution, magnitude and co-occurrence of design smells across desktop and mobile applications. We identified and analyzed fifteen (15) most frequent design smells across twelve (12) open-source object-oriented Java projects extracted from GitHub. This paper makes the following contributions:

1) We present a comparison of occurrence of design smells in desktop and mobile application with a focus on diversity, distribution, magnitude and co-occurrence using a combination of heuristics, statistics and unsupervised learning techniques.
2) We show that the aforementioned clustering approach can be leveraged to expose hidden and non innate relationships among design smells.
3) We discuss the implications of our study for researchers, software industry and on the development of design smell detection tools.

The rest of the paper is structured as follows: In section

II we explain key concepts of design smells, their effect on software maintainability and discuss related work. Next, in section III, we provide a detailed explanation of the dataset and methods used to conduct our study. Then, we present the results and discussions in section IV. We end with discussing implications in V, conclusions in VII and direction of future work.

II. Related Work

A. Design Smells

Design smells, also known as “anti-patterns" [7], “code smells" [8] or “bad smells", are indicators of issues in source code that can negatively affect maintainability of a software system as well as various programming activities [9]. Technically, code smells do not stop the system from functioning but can easily affect the development process, weaken the sustainability of software and increase the probability of its failure [10]. The existence of design smells in source code calls for code refactoring which is a common programming task aimed at improving the internal structure of existing software code without affecting its observable behavior. It greatly enhance software maintainability and generate a more manageable internal architecture [10]. An earlier study by Fowler [1] outlines 22 design smells and their corresponding refactoring techniques. Those design smells are programming language-independent but mostly targets object-oriented paradigm.

B. Design Smell Detection Techniques

The detection of harmful code smells which deteriorate the software quality has attracted a lot of research interests, both from academic and software industry. As such, significant studies towards effective detection of code smells have been conducted over the last few years. Sharma and Spinellis [11] grouped design smells detection strategies in five broad categories, which include; Metrics-based, Rules/Heuristic-based, History-based, Machine learning-based and Optimization-based smell detection.

1) Metric-based design smell detection: Metrics-based is the most common approach of design smells detection [12]. It is relatively easy to implement and normally follows three generic steps; (1) take source code as input and prepare a source code model such as Abstract Syntax Tree (AST). (2) detect a set of source code metrics that capture the characteristics of smells. (3) detect smells by a suitable threshold value.

2) Machine learning-based design smell detection: Several authors have applied machine learning in the detection of code/design smells with a common focus on supervised learning [6], [13]–[15] and cluster-based approaches [3].

Liu *et al.* [14] proposed a deep learning-based approach to detect Feature Envy. Their approach relies on both structural and lexical information. Labeled samples were automatically generated from open-source applications. The gold dataset is fed into two convolutional neural network layers and one fully-connected layer to perform classification. More recently, Barbez *et al.* [6] extended the approaches in [13] and [14] using ensemble machine learning method called *SMart Aggregation of Anti-patterns Detectors (SMAD)* to detect anti-patterns. SMAD was designed by intergrating several detection tools based on their internal detection rules to produce an improved prediction from a reasonable number of training examples. SMAD significantly outperformed other ensemble methods especially for the detection of two well-known anti-patterns i.e. God Class and Feature Envy in eight Java projects.

Although machine learning-based smell detection has demonstrated a lot of potentials as recorded by different authors, it is heavily dependent on a large amount of training dataset and availability of such dataset is still a huge concern.

3) Optimization: Most studies in this category utilize optimization algorithms such as genetic algorithms to detect anti-patterns in a software system. Saranya *et al.* [5] applied a genetic algorithm for model-level code smell detection. The motivation of their study was based on the limitation of rule and metrics-based code smell detection approaches. Specifically, defining the rules and identifying the correct threshold value in rule-based code smells detection is a tedious task and normally achieved through trial and error method. To address this issue, this work introduced a Euclidean distance-based Genetic Algorithm and Particle Swarm Optimization (EGAPSO). The result of EGAPSO proves to be effective when compared to other code smell detection approaches such as DEtection & CORrection (DECOR) [16].

C. Impact of Design Smells on Software Maintenance

It is important to note that code smells are not errors and therefore, they do not prevent the software from functioning. In some circumstances, code smells are introduced by developers while implementing important patches or developing a “hack" or “workaround" as a sub-optimal solution to existing problem [2]. According to previous studies, the existence of design smells makes the program complex to comprehend, summarize and modify [3], [17], which negatively affect software quality [4].

Soh *et al.* [9] studied the effects of code smells at the developer's activity level i.e. code reading, editing, searching, and navigating. The experiment involved six expert developers performing maintenance tasks on four Java applications. Each developer performed two tasks while logs were monitored. An annotation schema was then defined to identify developers' activities and assess whether code smells affect his/her different maintenance task. Result of their study indicated that code smells indeed affect the effort of certain kind of activities but the effect is contingent on the type of maintenance.

It is also noted that design smells are associated with the occurrence of software bugs, which affect maintenance tasks. Cairo *et al.* [18] carried out a systematic literature review on published studies that provide evidence of the influence of code smells on the occurrence of software bugs. In their study, 24 code smells were found to be more influential in the occurrence of bugs. Specifically, *God Class, Shotgun Surgery* and *God Method* were found to be significant contributors and positively associated with error proneness.

D. Design Smells in Desktop and Mobile Applications

A few previous papers carried out work closely related to the study of design smells in desktop and mobile applications. To begin with, Mannan *et al.* [19] conducted a study to understand code smells in android applications and how they compare with desktop application. Their study involved a large corpus of desktop and android application collected from Github.

Habchi *et al.* [20] studied code smells in iOS by analyzing 279 open-source iOS apps. In this paper, the authors considered the presence of object-oriented and iOS-specific code smells by analyzing 279 open-source iOS apps. Source code was analyzed by extending PAPRIKA toolkit in order to accommodate the detection of code smells in Objective-C or Swift language. Their observation shows 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.

Another interesting replication study by Palomba *et al.* [21] focused on investigating code-smells co-occurrence using association rule. This study was carried out on a dataset composed of 395 releases of 30 software systems, capturing 13 code-smells. Their results highlighted some expected relationships but also revealed some co-occurrences missed by previous research.

Despite the results obtained from these study, the following extensions can be made; The study by Mannan *et al.* [19] focused mainly on code smells which tends to affect readability and simplicity. The task of refactoring in this case involves renaming or extracting to methods. Design smells, on the other hand, tend to be more subtle. They usually affect maintainability and flexibility. Next, their study [19] did not look at the general co-occurrence of code smells and how these co-occurrences varies across desktop and android application. Palomba *et al.* [21] focused on general co-occurrence of code smells in Java but not in mobile applications. Moreover, that projects selected in this study comprise a mixer of Java desktop applications and libraries, whose internal implementation slightly differs.

III. Method

In this section we discuss the approach taken to conduct this study including data collection, preprocessing and analysis techniques.

A. The Dataset

Our dataset is based on twelve (12) real-life open-source Java projects mined from GitHub. Seven (7) of the projects are mobile (android-based) applications and the other five (5) are desktop applications. The android projects were selected from a list of projects previously studied by Mannan *et al.* [19] found here[1]. For our study, we focused on the latest releases from GitHub. The selection criteria for desktop applications was based on two significant characteristics:

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

- All projects rely on the Java Swing library for GUI design.
- Cross-platform compatibility (i.e. can function on both Windows, MacOS and Linux operation system) and depended on Java Core libraries for the design of its backend logic.

The selected projects constitute a total of 1,601,369 Java Lines of Code (LoC). We choose Java-based projects because they account for a wide variety of open source projects hosted on different code repositories and at the time of this research, Java was among the top 3 popular programming languages within the software industry. Moreover, Java is used by billions of devices across the globe. The details of the selected projects are shown in Table I.

TABLE I: Projects selected in this study including total LoC and number of Design Smells (#DS) in each codebase.

No.	Domain	Project	Version	#LoC	#DS
1	Desktop	SweetHome3d	5.6	104,059	206
2	Desktop	Mars Simulation	3.1.0	255,459	875
3	Desktop	ArgoUML	0.35.1	177,372	1,160
4	Desktop	JEdit	5.5.0	124,164	605
5	Desktop	GanttProject	2.9.11	66,709	394
6	Mobile	K9 Mail	5.600	93,540	247
7	Mobile	Bitcoin Wallet	6.31	18,079	50
8	Mobile	KeepassDroid	2.5.9	17,916	156
9	Mobile	Opentrip Planner	2.1.5	9,760	28
10	Mobile	Telegram	6.1.1	541,694	540
11	Mobile	Tweet Lanes	1.4.1	25,886	105
12	Mobile	Text Secure	4.69.5	166,731	799
	Total			**1,601,369**	—

B. Data Preprocessing

For the preprocessing task, we passed the project class files as input to an anti-pattern detection tool. Particularly, we used *Pattern Trace Identification, Detection, and Enhancement in Java* (Ptidej) tool. It is a open source Java-based reverse engineering tool suite that includes several identification algorithms for idioms, micro-patterns, design patterns, and design defects [22]. Using this tool, we were able to detect and select fifteen (15) frequent anti-patterns across the selected projects which includes: *LongMethod, ComplexClass, LongParameterList, BaseClassShouldBeAbstract, SpeculativeGenerality, ClassDataShouldBePrivate, ManyFieldAttributesButNotComplex, MessageChain, SpaghettiCode, RefusedParentBequest, SwissArmyKnife, Blob, AntiSingleton, LargeClass, LazyClass*. Design smells are detected and stored in *".ini"* files. The file names are tagged with a specific design smell type. For example, in the k-9 mail project, **AntiSingleton** design smell is stored as *"DetectionResults in K9 for **AntiSingleton.ini**"*. Our goal is to extract class names and the corresponding design smell detected in that class.

We apply heuristics to determine the structure and pattern of class names in the detected anti-pattern result files for the different projects. We use python regular expressions to extract class names and associated them with respective design smell type. A value of **1** was assigned if a particular anti-pattern is detected in a class otherwise **0** is assigned. Table II shows a sample of the final output of the preprocessing tasks.

C. Data Analysis

We start by grouping the data to create a collection of aggregated number of each design smells in specific project. Next, we grouped the data according to whether the code-base is a mobile (android) app or desktop software as shown in Table III.

D. Clustering

Clustering is one of the most important concepts for unsupervised learning in machine learning. In this study, we used Powered Outer Probabilistic Clustering (POPC) [23]. The choice of this algorithm was based on the following motivation: First, numerous clustering algorithms including the popular k-means algorithm, require the number of clusters to be specified in advance which is a huge drawback. Some studies use the silhouette coefficient, elbow method, and other approaches to determine the optimal number of clusters. However, those methods have their limitations, for example: sometimes the elbow method fails to give a clear “elbow point”. Second, k-means is not very suitable for a binary feature sets.

Using POPC, we do not need to specify the number of clusters upfront. It tries to mitigate these drawbacks using back-propagation. The algorithm is observed to work quite well on binary datasets and converges to the expected (optimal) number of clusters on theoretical examples as elaborated by Taraba [23].

Based on the processed dataset in Table II, we constructed two different datasets for the task of clustering. The first dataset consist of desktop data while the second contain mobile data. This was carried out to determine if there were any observable differences in the cluster formation across the datasets. The output of POPC clustering was used to group design smells based on their occurrence in each set of data as shown in Figure 4, using the following procedure:

1) First, we constructed a table of clusters and design smells for both desktop and android- Table IV.
    - For each cluster (c) of classes, we compute the total number of each design smell.
    - If the total number of a given design smells is > 0, we assign a value 1 in its row, otherwise, we assign a value 0. The output of this operation is shown in Table IV.
    - We repeat this process for all the clusters.
2) Secondly, we extract the design smell rows and create a binary matrix. This matrix is treated as an n-dimensional array which is then passed to a dendrogram creation function. We use the python Plotly package which performs hierarchical clustering on data and represents the resulting tree.

IV. Results

In this section, we present and discussion the results of our study. We answer the following research questions:

A. RQ: Does the type of application i.e. desktop or mobile influence the variations in diversity, distribution and magnitude of design smells occurrence? If so, are these variations statistically significant?

To answer this research question, we start by discussing some of the key differences and similarities between mobile and desktop applications. According to the dataset, both mobile and desktop projects are based on Java programming language and fundamentally obey the OOP programming paradigm. However, some key notable differences exist, for example, desktop applications mostly rely on the Java Swing library for Graphical User Interface (GUI) design while mobile (android) applications are based on XML as their underlying language for the design of GUI [19]. Table V presents some of these key notable differences. We believe that these differences could influence the diversity, distribution and magnitude of design smell occurrence.

1) Diversity of Design Smells: We found some interesting variations in the variety of design smells that occur in desktop and mobile applications. Generally, we observed that desktop applications have a diverse type of design smells compared to mobile applications. For example, looking at Table III, we can observe that desktop applications account for up to 93% of the total type of design smells detected in the entire data set whereas mobile applications takes up about 73%. These variations are caused by the following design smells: *RefusedParentBequest, SpaghettiCode, MessageChains and SwissArmyKnife* that occurred in desktop applications only while *LargeClass* was observed in only the mobile applications.

We think these variations can be explained based on the differences in the workflow of android and desktop applications (as highlighted in Table V). Based on these differences, we expect more types of design smells to occur in desktop than in mobile applications. For example, android applications are built upon the android frameworks which encapsulates low-level functionalities of the Android OS. Thus, the developer does not have to implement several classes at UI and activity management level, thereby reducing the possibility of inducing design smells

2) Distribution and Magnitude of Design Smells: The second part of this research question focus on understanding the distribution and quantity of design smells in desktop and mobile applications. Figure 1 shows both the distribution and magnitude of design smells across desktop and mobile applications. We noticed that design smells are almost always more in desktop applications than in mobile applications. This difference can be observed in Figure 2 where desktop and mobile applications account for 67.5% and 32.5% of the total number of design smells in our corpus respectively. A rather large variation exists in the quantity of *Blob* and *Refused-ParentBequest* in desktop and mobile applications. Next, we explored the relationship between lines of code (LoC) and magnitude of design smells in desktop and mobile application. Figure 3 shows a scatter plot of total LoC against magnitude

TABLE II: Sample output of processed design smell files.

No.	FullClassPath	LongMethod	LazyClass	Blob	ComplexClass	...
1	k9mail.src.main.java.com.fsck.k9.controller.SimpleMessagingListener	1	1	1	0	...
2	org.thoughtcrime.securesms.mms.AudioSlide	1	0	0	0	...
3	org.telegram.ui.Components.GroupCreateSpan	1	0	0	1	...
4	org.telegram.ui.Cells.TextSelectionHelper	1	0	0	1	...
5	com.keepassdroid.database.PwDate	1	0	0	1	...
6	com.tweetlanes.android.core.view.HomeActivity	1	0	0	1	...

TABLE III: Diversity of design smells across desktop and mobile applications.

No.	**Design Smells**	**Desktop**	**Mobile**
1	LongMethod	✓	✓
2	ComplexClass	✓	✓
3	LongParameterList	✓	✓
4	LazyClass	✓	✓
5	Blob	✓	✓
6	ClassDataShouldBePrivate	✓	✓
7	RefusedParentBequest	✓	✗
8	AntiSingleton	✓	✓
9	BaseClassShouldBeAbstract	✓	✓
10	SpeculativeGenerality	✓	✓
11	SpaghettiCode	✓	✗
12	ManyFieldAttributesButNotComplex	✓	✓
13	MessageChains	✓	✗
14	SwissArmyKnife	✓	✗
15	LargeClass	✗	✓

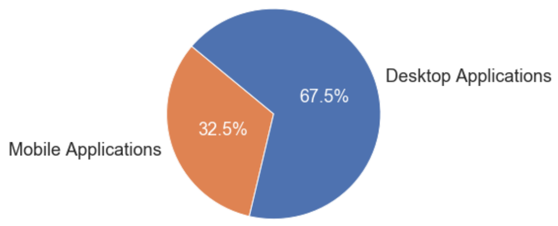


Fig. 2: Aggregated percentages of design smells across desktop and mobile applications.

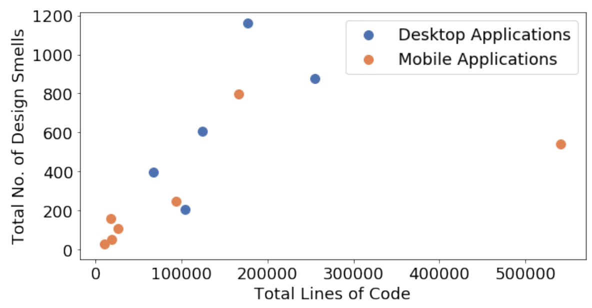


Fig. 3: Comparison of the lines of code with magnitude of design smells.

for each selected project within the two ecosystems. We found that there is an observable linear correlation between LoC and magnitude of design smell. Particularly, design smells tend to increase proportionally to the increase in project size, except for a few cases. We think this result is interesting and worth further investigation especially using various project releases.

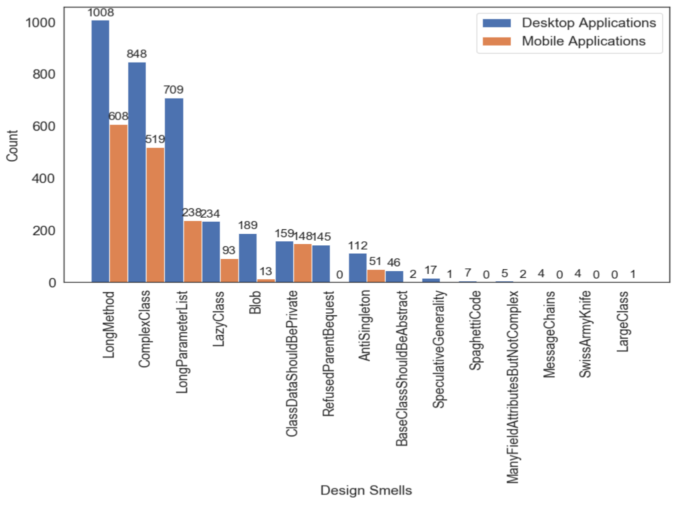


Fig. 1: Comparison of the distribution of design smells in desktop and mobile applications.

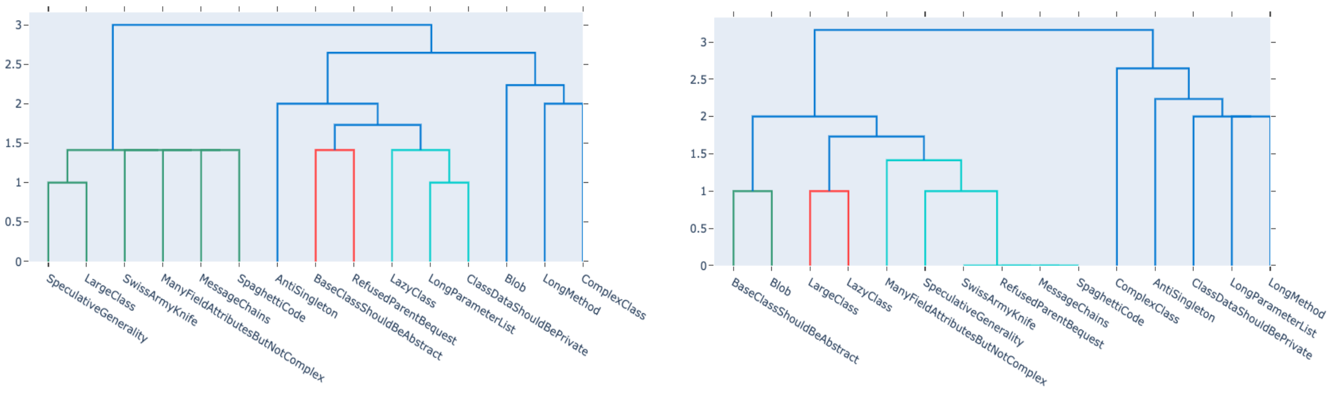
We also went further to study the distribution of design smells using POPC clustering algorithm discussed in section III. The motivation was to understand design smell distribution from an unsupervised learning perspective. This way, we can observe pairs or groups of design smells that often occur together and/or have a similar characteristics in desktop versus mobile applications from the dataset.

The clustering results in Figure 4 reveal groups of design smells that often co-occur in desktop or android applications. Some of the clusters are expected while others are not obvious and call for more study to understand the reason for their appearance and relationships. For example; *SpeculativeGenerality* and *SwissArmyKinfe* both occur in the same cluster for desktop and mobile applications. This is expected because these design smells are theoretically related. Other similar relationships in the clusters produced by desktop and mobile data include: *ComplexClass* and *LongMethod*, *ManyFieldAttributesButNotComplex, MessageChains, SpaghettiCode*.

3) Statistical Significance: To determine whether the variations in diversity, distribution and magnitude of design smell across desktop and mobile applications is statistically significant, we conducted a statistical test using the Welch's two-sample t-test. The Welch's t-test is a less restrictive version of the student's t-test and mostly recommended for dealing with data of unequal variance and sample size, while maintaining the normality assumption. We choose this particular test because the number of projects we selected for desktop and

TABLE IV: An example of data constructed from the output of POPC clustering to create the Dendrograms in Figure 4.

No.	DS	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
1	LongMethod	0	0	1	0	0	0	0	0	0	0
2	ComplexClass	0	0	0	0	1	0	0	0	0	0
3	LongParameterList	1	1	1	1	0	0	1	0	0	0
4	LazyClass	0	0	0	1	0	0	0	0	0	0
5	Blob	1	1	1	0	0	1	1	1	0	0
6	ClassDataShouldBePrivate	1	1	1	0	1	0	1	1	0	1
7	RefusedParentBequest	1	1	1	1	0	1	1	0	0	0
8	AntiSingleton	0	0	0	0	0	1	0	0	0	0
9	BaseClassShouldBeAbstract	1	1	1	0	0	1	0	0	0	0
10	SpeculativeGenerality	0	0	0	0	0	0	0	0	1	0
11	SpaghettiCode	0	0	1	0	1	0	1	0	1	0
12	ManyFieldAttributesButNotComplex	1	0	0	1	0	1	1	0	0	0
13	MessageChains	1	1	1	1	1	1	1	1	1	0
14	SwissArmyKnife	0	0	0	0	0	0	0	0	0	0
15	LargeClass	1	1	1	0	0	0	1	0	0	0



(a) Desktop

(b) Mobile

Fig. 4: Dendrogram showing groups of design smells that co-occur frequently.

mobile applications is not the same. We carried out this test using the data in Figure 1 and obtained the following results: **Welch's test: -1.201, p-value: 0.242**. The result indicates that, despite the various observable variations in diversity, distribution and magnitude of design smell across desktop and mobile applications, these variations are not statistically significant. Therefore, we can not conclude that design smells often occur more frequent in desktop or mobile applications. This result is also consistent with previous papers focusing on code smells, for example, the paper by Mannan *et al.* [19].

V. Implications

In this section, we present the implications of this study to the researchers, software engineers and on the development of design smell detection tools.

A. To Researchers

Despite the variations in diversity, distribution and magnitude of design smell between desktop and mobile, our result indicates that almost all instances of design smells are well represented in both software domains as shown in Figure 1. Therefore, researchers studying design smells in one domain have a high possibility of obtaining a representative set of data which can easily be generalized for both domains. This is also backed by our statistical significance test result which indicates that there is no statistical difference in the number of design smells across desktop and mobile applications.

The clustering results in Figure 4. provides some practical confirmation of theories related to shared characteristics and similarities among design smells. For example, we were able to show, using unsupervised learning that *Speculative Generality* and *SwissArmKnife* are closely related. However, we also found some unexpected relationship/similarities in the clusters which require more research to understand them and make recommendations. We, therefore, encourage researchers to consider exploring this direction in future studies.

Our study also provides a good ground for software educators to demonstrate various design principles to students. As such, learners can practically observe examples of well-designed and poorly designed systems for a wide range of software systems.

B. To Software Developers

We discuss three significant implications of this study to software developers as follows:

TABLE V: Notable differences between Java-based desktop and Android applications.

No.	Desktop Application	Android Application
1	Application entry point is dependent on the existence of a special method i.e. the main method.	There is no main method when developing mobile applications. The entry points are given by event-handlers such as *onCreate, onPause, onResume*, etc
2	Application's underlying GUI is designed using Java Swing library (core Java language)	There is another layer of abstraction i.e. complete separation of the application logic from its presentation. Moreover, the GUI is constructed using eXtensible Markup Language (XML).
3	Consist of all J2SE libraries, Swing and JavaFX, etc.	Android does not have all J2SE APIs, Swing or JavaFX.
4	It solely depends on OOP paradigm	Although based on OOP, the android mobile apps employ reactive, event-driven programming paradigm
5	The application directly benefits from the host infrastructure, which can be easily scaled vertically or horizontally	Designed with resource limits in mind. Some of the app's capabilities are constrained by the underlying hardware infrastructures such as memory, storage, processing power and peripherals.
6	Source code is compiled to Java Bytecode	Source code is compiled to java bytecode then to DEX bytecode (two stages)

1) Software Design and Development: As shown in Figure 1, developers can know from the start of any new software project that they should pay attention to specific implementation details of their application to mitigate common design smells. For example; they can observe that *LongMethod, ComplexClass* and *LongParamaterList* is most likely to occur in an application. We also show groups of design smells that frequently co-occur. These knowledge can help developers to correctly plan their implementation and/or provide guidelines for contributors to mitigate these design smells, thereby limiting future software failure due to sloppy or unintended programming/implementation choices.

2) Quality Assurance: Software Quality Assurance (SQA) is an essential aspect of software engineering that involves processes and methods to ensure proper software quality such as conformance to standards or models. This study provides evidence to developers and quality assurance personnel of the importance of design smell analysis in assessing the quality of their systems by showing the diversity, distribution and magnitude of design smell which can negatively impact software maintenance effort.

3) Guided Code Review and Refactoring: Code review and refactoring are common exercises carried out by developers to (i) ensure code quality and (ii) improving the internal structure of existing software code without affecting its observable behavior [10]. However, the cost of reviewing and refactoring source code becomes expensive in terms of time and resource, especially for evolving software systems. Therefore, there is a need for the simplification of those processes. As such, we believe that our study results can guide developers by quickly pointing them to specific features of a source code that often result in poor software quality such as *long method*, *complex class* or *long parameter list*.

C. On The Development of Design Smell Detection Tools

Design smell detection tools are significant not only for research purposes but also ensuring high-quality software design. The good news is that judging from Figure 1, detection tools developed for desktop application will probably always work for android applications as well. However, we believe that further improvements such as metrics optimization and enhancing code linting can significantly boost design smell detection tools as discussed below.

1) Metrics Optimization: Design smells are detected using a combination of software metrics. However, Metrics-based smell detection method has some known limitations such as its inability to detect many smells using only commonly known metrics. Besides, metrics-based strategies heavily depend on the choice of best threshold value by the researcher, which is normally a significant challenge since this choice is almost always empirical and trial-and-error [11]. This research provides an opportunity for design smell detection tool developers to review those metrics and tailor them for the detection of specific design smell or combination of design smells based on the way they occur across desktop and android application. Moreover, developers can use the knowledge in Figure 4 to optimize design smell detection tools to become more efficient through the use of just a few metrics to detect a combination of design smells.

2) Improve Code Linting: A large percentage of software engineers (both junior and senior) embrace the use of code linters in their daily development activity. A linter analyzes source code to detect flaws, check style conventions, potential bugs and other code constructs [24]. However, most linters cannot flag design smells. We believe that the results of this study can motivate design smell detection tool builders to integrate design smell detection capability in linters as an extension or plugin.

VI. Threats to Validity

A. Construct Validity

Our goal was to compare the occurrence of design smells in desktop and mobile applications. We believe that we were able to achieve this goal by comparing the two groups in regards to diversity, distribution, magnitude and co-occurrence of design smells.

B. Internal Validity

In this study, we realized solely on *ptidej* tool suite for the detection of design smells. Therefore, the accuracy of our results also depends on the accuracy of this tool. However, the efficacy of *ptidej* has been evaluated in previous study [25]. Besides, *ptidej* tool suite is freely available and able to detect a large number of design smells.

We carried out our analysis on just a single version of each selected project. It is possible that the results can vary if historical data is considered. However, since our focus was not on the analysis of software change history, we consider this threat acceptable and an opportunity for future work.

C. External Validity

Regarding the generalizability of our result, first, we are aware that we carried out this study on Java-based application only. Other platforms that use OOP languages exist for example; windows mobile based on C#, Apple iOS based on Objective-C/Swift, etc. However, we believe that the methods used in this study can be generalized to other OOP systems in various programming languages because the principle of OOP is consistent regardless of the implementing programming language.

The dataset used in this study was generated from only 12 GitHub projects. Although we believe that the size of our dataset is considerable, using a larger-sized dataset would give more confidence to the results presented in this paper.

VII. Conclusion and Future Work

In this paper, we conducted an exploratory study to compare design smells in desktop and mobile application using a sizable dataset of twelve (12) Java-based open-source projects. We reported empirical evidence on the variations in diversity, distribution, magnitude and co-occurrence of design smells using statistical methods and unsupervised learning. The result of the study indicated that desktop and mobile application are quite similar in term of design smell occurrence. We also found pairs/groups of design smells that often co-occur. Some of the pairs/groups are expected (e.g. *SpeculativeGenerality, SwissArmyKinfe*), while others (e.g. *LongParameterList, ClassDataShouldBePrivate*) require further study to understand any innate relationships.

We plan to extend the study to include class role-stereotypes [26]. It is quite intuitive that both design smell and role-stereotype play major roles in the design and maintenance of a software system. It would be interesting to see how design smells vary across role-stereotypes in desktop and android application. We are also interested in understanding the variation of design smells in cloud-native versus traditional applications. We find this important because numerous software development activity has now shifted to the cloud.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999, ISBN: 0-201-48567-2.

[2] A. Kaur, "A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes," *Archives of Computational Methods in Engineering*, pp. 1–30, 2019.

[3] A. Imran, "Design smell detection and analysis for open source java software," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, pp. 644–648.

[4] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?" In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2018, pp. 612–621.

[5] G. Saranya, H. K. Nehemiah, A. Kannan, and V. Nithya, "Model level code smell detection using egapso based on similarity measures," *Alexandria engineering journal*, vol. 57, no. 3, pp. 1631–1642, 2018.

[6] A. Barbez, F. Khomh, and Y.-G. Guéhéneuc, "A machine-learning based ensemble method for anti-patterns detection," *Journal of Systems and Software*, vol. 161, p. 110 486, 2020.

[7] S. Kaur and S. Singh, "Influence of anti-patterns on software maintenance: A review," *International Journal of Computer Applications*, vol. 975, p. 8887, 2015.

[8] T. Guggulothu and S. A. Moiz, "An approach to suggest code smell order for refactoring," in *International Conference on Emerging Technologies in Computer Engineering*, Springer, 2019, pp. 250–260.

[9] Z. Soh, A. Yamashita, F. Khomh, and Y.-G. Guéhéneuc, "Do code smells impact the effort of different maintenance programming activities?" In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, vol. 1, 2016, pp. 393–402.

[10] B. Turkistani and Y. Liu, "Reducing the large class code smell by applying design patterns," in *2019 IEEE International Conference on Electro Information Technology (EIT)*, IEEE, 2019, pp. 590–595.

[11] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.

[12] B. Bafandeh Mayvan, A. Rasoolzadegan, and A. Javan Jafari, "Bad smell detection using quality metrics and refactoring opportunities," *Journal of Software: Evolution and Process*, e2255, 2020.

[13] A. Kaur and S. Singh, "Detecting software bad smells from software design patterns using machine learning algorithms," *International Journal of Applied Engineering Research*, vol. 13, no. 11, pp. 10 005–10 010, 2018.

[14] H. Liu, Z. Xu, and Y. Zou, "Deep learning based feature envy detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 385–396.

[15] Y. Yin, Q. Su, and L. Liu, "Software smell detection based on machine learning and its empirical study," in *Second Target Recognition and Artificial Intelligence Summit Forum*, International Society for Optics and Photonics, vol. 11427, 2020, 114270P.

[16] 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," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.

[17] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

[18] A. S. Cairo, G. d. F. Carneiro, and M. P. Monteiro, "The impact of code smells on software bugs: A systematic literature review," *Information*, vol. 9, no. 11, p. 273, 2018.

[19] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in android applications," in *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, IEEE, 2016, pp. 225–236.

[20] S. Habchi, G. Hecht, R. Rouvoy, and N. Moha, "Code smells in ios apps: How do they compare to android?" In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, IEEE, 2017, pp. 110–121.

[21] F. Palomba, R. Oliveto, and A. De Lucia, "Investigating code smell co-occurrences using association rule learning: A replicated study," in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, IEEE, 2017, pp. 8–13.

[22] Y.-G. Guéhéneuc, "Ptidej: A flexible reverse engineering tool suite," in *2007 IEEE International Conference on Software Maintenance*, IEEE, 2007, pp. 529–530.

[23] P. Taraba, "Clustering for binary featured datasets," in *The World Congress on Engineering and Computer Science*, Springer, 2017, pp. 127–142.

[24] S. Habchi, X. Blanc, and R. Rouvoy, "On adopting linters to deal with performance concerns in android apps," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2018, pp. 6–16.

[25] G. Rasool, P. Maeder, and I. Philippow, "Evaluation of design pattern recovery tools," *Procedia Computer Science*, vol. 3, pp. 813–819, 2011.

[26] R. J. Wirfs-Brock, "Characterizing classes," *IEEE software*, vol. 23, no. 2, pp. 9–11, 2006.