Information Systems Frontiers (2022) 24:659–676
https://doi.org/10.1007/s10796-020-10100-w

On the Impact of Aesthetic Defects on the Maintainability of Mobile Graphical User Interfaces: An Empirical Study

Makram Soui[1] · Mabrouka Chouchane[2] · Narjes Bessghaier[3] · Mohamed Wiem Mkaouer[4] · Marouane Kessentini[5]

Accepted: 25 December 2020 / Published online: 20 February 2021

Abstract

As the development of Android mobile applications continues to grow and to follow up its high increase in demand and market share, there is a need for automating the evaluation of Graphical Mobile User Interfaces (GMUI) to detect any associated defects as they are perceived to lead to bad overall usability. Although, there is growth in research targeting the assessment of mobile user interfaces, there is a lack of studies assessing their impact on quality. The goal of this work is to analyze the impact of defects on the maintainability of user interfaces by studying the connection between the existence of defects and the change-proneness of user interfaces. We empirically study the impact of 8 aesthetics defects in 56 releases of 5 Android applications and examine the diffuseness of GMUI defects throughout mobile apps evolution. Then, we investigate whether infected classes are changed more frequently, and have a larger change-size than other non-infected classes in terms of Change Frequency (CF) and Change-Size (CS). Moreover, we studied the survivability and co-occurrences of GMUI defects in order to prioritize their corrections. Our empirical validation confirms that the infected user interfaces are more prone to undergo many changes than other user interfaces, and there are some severe aesthetic defects still exists even after makingmany improvements in the code that may need more maintenance efforts.

Keywords Aesthetics defects · Change-size · Correlation · Evolution of Android GMUI

✉ Makram Soui
m.soui@seu.edu.sa

Mabrouka Chouchane
chouchane.mabrouka@univgb.rnu.tn

Narjes Bessghaier
narjes.bessghaier.1@ens.etsmtl.ca

Mohamed Wiem Mkaouer
mwmvse@rit.edu

Marouane Kessentini
marouane@umich.edu

[1] College of Computing and Informatics Saudi Electronic University, Riyadh, Saudi Arabia

[2] National School of Computer science of Manouba, Manouba, Tunisia

[3] Ecole de Technologie Superiere (ETS), Quebec, Canada

[4] Rochester institute of technology, Rochester, NY, USA

[5] University of Michigan, Ann Arbor, MI, USA

1 Introduction

With the evolution of smartphones, mobile applications (apps) are becoming one of the pillars of software market (research 2013). Nowadays, industry heavily relies on mobile apps to reach end-users swiftly and smoothly. Nearly 197 billion apps were downloaded in 2017 (Statista 2020), and Android apps have been leading the market share with 87% in 2016 (Store 2020). One of the key features standing behind the exponential usage growth of mobile apps is their usability (Paiano et al. 2013). Mobile apps are more user-centered by a trade-off between providing an interactive and appealing design along with high-performance execution. Henceforth, the longevity of the app in the market requires finding the right compromise between continuously optimizing its features while maintaining its current performance. While mobile clients typically can choose between various apps providing similar services, developers, on the other hand, are facing the challenge of maintaining the high quality of their app's design while rapidly evolving it with newly introduced features to guarantee their maintainability and competitiveness.

The tremendous amount of changes, introduced while evolving the app, is responsible for the deterioration of its code design. These bad design decisions are known as code smells (Fowler and Beck 1999), and their existence negatively impacts the understanding and maintenance of the app's codebase (Yamashita and Moonen 2013; Lanza and Marinescu 2007). These code smells can be classified into two categories: external and internal. Internal smells are obtained from the source code analysis, and indicate poor design decisions. As for external smells , they are symptoms of poor usability choices, eventually experienced at the GMUI level. Thereby, the good design of mobile Graphical User Interfaces (GMUI) plays an important role in promoting the quality of the app.

Aesthetics requirements are the user interface or the end appearance of the application. Most of the time it keeps changing between different versions. This happens especially when the end-users demand a new set of requirements or complain about design choices. As it generally happens, the clients detect and request changes in the User Interface (UI). Aesthetics visual design aims at improving the usability of the application and its maintainability for business purposes. The need for a high visual attractiveness of the GMUI is essential for the end-users, so that the interaction of the application becomes very simple and effective. In order to repair an issue with the GMUI, clients will report the problem and send their feedback to the development group through the Play stores where the apps are published. Developers are mostly concerned with the optimization of the functional aspect of the application than the GMUI design aspect as it is mostly rated as low negative impact factor. However, GMUI's code may achieve up to 45% of the total application (Myers 1995), and though they cannot create a direct impact on the apps performance, they still can create problems with the usability and productiveness among the users. Nevertheless the critical impact of functional bugs than that of aesthetics defects, the latter still be a priority for end-users.

When several complaints are being received from end-users, developers will try to meet these new requirements. And it is here where survivability points will be affected to each detected defect to prioritize its fixing. However, the question that arises here is: do developers always succeed in meeting users needs and reduce the number of design defects. A good beginning to such an endeavor is to: (1) control the evolution pace of aesthetics defects all throughout apps evolution. We aim at getting an insight over the quality of performed code changes according to the number of defects in release r(k) and its subsequent r(k+1). (2) investigate whether infected classes are changed more frequently, and have a larger change-size than other non-infected classes in terms of Change Frequency (CF) and change-size (CS). (3) study the types of aesthetics defects requiring more maintenance effort, mainly by studying their survivability and co-occurrence in the studied mobile apps.

Our main findings prove that infected GMUI source files experience an increase in the change-size in comparison with the non infected ones. In addition, we find that there is some defects that tend to be more persistent than other defects. Furthermore, we notice the co-occurrence of some pairs of defects, which advocates prioritizing their correction with respect to survivability.

The remainder of this paper is organized as follows: Section 2 defines the related work of this research. Section 3 introduce the background. Section 4 presents our empirical study and main research questions while Section 5 provides the empirical validation. Section 6 discusses the key findings of the experiments. Section 7 presents the potential threats to validity. The paper ends with Section 8 that concludes the empirical study and outlines our future directions.

2 Related Work

Since there is no consensus on how to fix and prioritize the detected structural aesthetics defects. Several studies have focused on estimating the survivability of internal and external code smells. These studies (Olbrich et al. 2010; Li and Shatnawi 2007; Khomh et al. 2009; Mkaouer et al. 2017; AlOmar et al. 2019) analyzed the correlation between code smells types and look for a possible cause-effect relationship by verifying whether removing a specific type of code smell results in reducing the system's proneness to changes, which also shrinks the maintenance overhead. In that vein, Olbrich et al. (2010) have empirically investigated the correlation between two smells God and Brain classes regarding their change frequency and change-size along with different releases. The findings show that class size made the God and Brain classes more subject to defects. Consequently, splitting functionalities over different classes will reduce the occurrence of code smells. Li and Shatnawi (2007), studied the correlation between code smells and a class error probability in three releases of Eclipse (3.0, 2.1, 2.0). The results showed that infected classes by Shotgun Surgery, God Class, and God Methods resulted in more class error probability than non-infected classes. Khomh et al. (2009) conducted an empirical analysis on 13 different versions of Azureus and Eclipse considering 9 code smells, to better understand the relationship between infected code classes and class change-size. The results validate that classes containing smells are more exposed to frequent updates than the remaining, and this observation holds across all versions and projects. They also observe a variance in change-proneness depending on the type of the smell. Tufano et al. (2016) aimed at determining the

developer's perception of test smells and came out with results showing that developers could not identify test smells very easily, thus, resulting in a need for automation. The results also showed that when a test code is committed to the repository that's the time when test smells are usually introduced. Bavota et al. (2012) conducted a human study and proved the strong negative impact of smells on test code understandability and maintainability. Another empirical investigation by the same authors (Bavota et al. 2015) indicated that there is a high diffusion of test smells in both open-source and industrial software systems, with 86% of JUnit tests exhibiting at least one test smell. The second study shows that test smells have a strong negative impact on program comprehension and maintenance. There are several existing works that focused on the evolution of code smells. In this context, Mercaldo et al. (2018) study the evolution of Android malware quality by measuring a set of software quality metrics that are divided into four groups (i.e. dimensional metrics, complexity metrics, object-oriented metrics, and Android-oriented metrics). The goal of this study is to define the evolution of such metrics in Android malware and goodware applications. In the same context, Gao et al. (2019) conduct a large-scale empirical study of the complex evolution of Android apps. This study used six metrics to measure the complexity of Android apps by presenting the impact of these applications on the maintainability process. While, Palomba et al. (2019) conducted a large-scale empirical study to analyze the influence of 9 Android-specific code smells on the energy consumption of 60 Android apps. The aim of this work is to determine the relationship between code smells and energy consumption.

These empirical studies highlight the importance for the community to develop tools to detect test smells and automatically refactor them. Refactoring code smells has been the focus of several studies (Mkaouer et al. 2014, 2015, 2017). Although external defects are detected at the GMUI level, they differ from the structural GMUI defects as it considers functional usability issues. External smells are produced through UI commands when a widget sends an event. The mending of these design smells requires rooting their cause in the java source code where the UI listeners are declared and associated. Blouin et al. (2017), detected the Blob listener smell by conducting a static code analysis procedure, and performed a refactoring operation by separating each command that composes a blob listener into a new UI listener applied on the same widget. To the best of our knowledge, no prior studies considered the impact of aesthetics defects on the maintenance of user interfaces. Pragmatically, are we deteriorating, maintaining or improving the UI structure as we modify the code? In fact, the Android UI layout is designed using Extensible Markup Language (XML) while Java is solicited for providing the core functionality. Therefore, the purpose of this paper is to help developers in the optimization of their GMUI maintenance activities by studying the diffuseness of aesthetics defects for the same UI throughout several releases. This study complements the existing effort of the community in reducing the impact of code smells and aesthetic defects.

3 Background

3.1 GMUI Aesthetic Defects

Although, the word design is mostly addressed to the functional part of the application, there is another fact as well beyond the functionality of a design: aesthetics, attractiveness and beauty (Norman 2004). A good mobile user interface (MUI) quality is compulsory to allow users to interact smoothly with the app. That is why a suitable design of the MUI is indispensable to increase the player loyalty towards an application. This latter is determined through the engagement scale, which includes 6 elements: appeal, novelty, focused attention, felt involvement, usability and durability (O'Brien and Toms 2010). In this study, we are focusing on the visual appearance of an interface and its effect on user satisfaction. As a matter of fact, aesthetics is considered an essential factor in perceived usability (Silvennoinen et al. 2014b). Consequently, it involves the user engagement level. An app can sustain in the market as long as it fulfills users needs with the provided functionalities. However, the importance of a good aesthetic design cannot be ignored. It is the visual attraction between a user and an app. Donald Norman, the UX expert, sees beauty as the momentum that forces us to buy products. Unfortunately, the role of visual elements in mobile applications is not intensely investigated compared to functionality. However, Silvennoinen et al. (2014a) has inspected how visual appearance influences the user experience in the mobile app context. Thus, aesthetics is seen as an element that attracts and engages the app users. Türkyilmaz et al. (2015), has investigated the importance of providing both aesthetics and functionality on websites interfaces by users opinion. The results show that aesthetics is as important as functionality and plays a significant role in user satisfaction. Code smells in previous studies are signs of bad design that are similar to our aesthetic defects, since both are carried symptoms of bad programming practices (Ines et al. 2017; Palomba et al. 2017; Kessentini and Ouni 2017). Code smells are detected in the back end of the system, while aesthetic defects are detected in the front end. We have detected eight aesthetic defects using our existing metrics-based plugin called PLAIN, which was published in a previous study (Soui et al. 2019). These aesthetic defects are presented in Table 1.

Table 1 List of aesthetics defects

Defects	Description	Abbrv.
Incorrect layout of widgets	It is related to the incorrect arrangement of MUI components. It concerns the alignment, dimension, orientation, depth and position of layouts.	(ILW)
Overloaded MUI	It is a bad density of MUI. In other words, users find the mobile interface too dense and so difficult to read.	(OM)
Complicated MUI	It is related to the MUI that includes too many widgets and features which cannot meet the users' needs.	(CM)
Incorrect data presentation	It is the incorrect extraction of information and their display on the mobile screen.	(IDP)
InCohesion of MUI	It is the lack of the interrelatedness of MUI components.	(ICM)
Difficult navigation	It is the lack of descriptive labels that can be used to define the additional information.	(DN)
Ineffective appearance of widgets	It occurs when MUI widgets follow an unexpected layout. It is related to the bad settings of the aesthetic aspect of a UI	(IAW)
Imbalance of MUI	It is an unequal distribution of the quantity of interactive objects of a given MUI.	(IM)

3.2 GMUI Metrics

GMUI evaluation is the process of measuring several interface properties, for the purpose of assessing its interaction with a user. Such evaluation is modeled by the construction of user's crossed interactions with the software, and the extent to which the user is satisfied with it (Akiki et al. 2014). The accuracy of such model heavily relies on the properties and measurements performed at the GUI level.

Aesthetics of mobile app is the visual attractiveness level of its user interface. It is related to structural and design beauty aspects (Moorthy and Bovik 2011). The evaluation of the aesthetics of graphical user interface is based on many characteristics such as quality of graphics, amount of text, and fonts, etc. In this context, Ngo et al. (2000) proposed a set of aesthetic measures for a graphical interface, such as balance, symmetry, equilibrium, sequence, order and complexity, cohesion, unity, proportion, simplicity, density, regularity, economy, homogeneity, and rhythm. The values of these measures can be calculated using the sizes and arrangements of components on the graphical user interface. Moreover, Hartmann et al. (2008) proposed several metrics such as usability, aesthetic, memory, overall preference, engagement, service, and information of the aesthetic attribute to enhance the visual equilibrium of label layout. Similarly, González et al. (2012) proposed many aesthetic metrics such as balance, linearity, and orthogonality. These three metrics aim to assess the graphical user interfaces aesthetically.

The metrics of structure aims to provide information about the quality of the mobile user interface design. Alemerien and Magel (2014) proposed several metrics of structure, such as alignment, balance, density, size and grouping. These metrics allow evaluators to assess the user interface structure. Sears (1993) also proposed a metric called layout appropriateness to organize widgets on the user interface. This metric takes the description of the sequence of widget actions as an input to calculate the cost of each sequence of these actions. Parush et al. (1998) developed a set of metrics: size, local density, alignment, and grouping to evaluate the screen layout of the graphical user interfaces. Furthermore, Constantine (1996) introduced a visual cohesion metric to assess the quality of user interface through a semantic aspect of widgets. Shoaib et al. (2011) proposed a metric of coherence of three cohesion modes: low, medium and high. In fact, this metric aims to evaluate the design quality of web applications. In this work, we use a set of evaluation metrics that were previously validated by Ngo et al. (2000). But, these metrics did not

take into account the characteristics of mobile devices such as the size of the screen. In a traditional desktop GUI, the values of metrics are calculated based on the area of layout but in the context of mobile computing, mobile apps can be either native, web-based, and hybrid (Masi et al. 2012). Furthermore, several studies highlighted the importance of visual aesthetic of mobile apps UI (Alemerien and Magel 2014, 2015). Typically, there exist two main categories of aesthetic user interface evaluation methods: qualitative and quantitative. The first category regroups subjective evaluation methods which aim to analyze how a design of user interface should respect a set of guidelines. The second category is based on automatic metrics-based tools. In this work, we used our tool PLAIN (Soui et al. 2017), which allows the measurement of eight aesthetic metrics inspired from Ngo et al. (2000) and described as follow:

- Regularity (RM) It measures the consistency spacing between all the MUI components.

$$RM = 1 - \left(\frac{N_{av} + N_{ah} + N_{sp}}{3n}\right) \quad (1)$$

Where: (N_{av}): numbers of vertical alignment points.
(N_{ah}):the numbers of horizontal alignment points.
(N_{sp}):the number of distinct distances between column and row.
n:the number of the components.

- Composition (COM) It counts the number of MUI components that are semantically linked in the same boundary.

$$COM = 1 - \left(\frac{G + UG}{2n}\right) \quad (2)$$

Where:
(G): number of groups with clear boundary.
(UG): number of ungrouped objects.
(n): number of the components.

- Sorting (SM) It aims to rank MUI components in a logical and sequential ordering according to the eye movement.

$$SM = 1 - \left(\frac{(\sum_{j=UL,UR,LL,LR}(q_j \sum_{i=1}^{n} Ni, j))}{4n}\right) \quad (3)$$

Where:
(UL): upper-left.
(UR): upper-right.
(LL): lower-left.
LR: lower-right.
($N_{i,j}$): is the number of object on the quadrant j.
q_{UL}=4, q_{UR}=3, q_{LL}=2, q_{LR}=1.

- Complexity (CM) It measures the complexity of interface by counting the number of rows and columns on the interface.

$$CM = \left(\frac{Nvap + Nhap}{2n}\right) \quad (4)$$

Where:
(N_vap)= number of vertical alignment points.
(N_hap)= number of horizontal alignment points.
(n) = number of objects.

- Integrality (IM) It count the number of different sizes of MUI components and the number of spacing be-tween it.

$$IM = 1 - \left(0.5\left[\frac{|N_{size} - 1|}{n} + \frac{|asc + \sum_i^n a_i|}{2a_{MUI}}\right]\right) \quad (5)$$

Where:
(N_{size}) =number of various sizes of objects.
(n) = the number of objects.
(a_{MUI})= area of MUI.
(a_{sc})= area of the screen.
(a_i) = area of the object i.

- Density (DN) It measures the total number of components in the MUI.

$$DM = 0.5\left|\frac{\sum_i^n a_i}{a_{MUI}} + \frac{a_{MUI}}{asc}\right| \quad (6)$$

Where:
(a_i)= area of object i.
(a_{sc})= area of the screen.
(n)= number of object.
(a_{MUI})= area of MUI.

- Symmetry (SYM) It aims to have an equal distribution of MUI components on the right and the left side of MUI.

$$SYM = 1 - \frac{|SYM_{vertical}| + |SYM_{horizontal}| + |SYM_{radial}|}{3} \quad (7)$$

Where: ($SYM_{vertical}$, $SYM_{horizontal}$, SYM_{radial}) are, respectively the vertical, horizontal and radial symmetries.

- Repartition (RM) It refers to the distribution of the components on the mobile user interface.

$$RM = \frac{(\frac{n}{4}!)^4}{n_{UL}!n_{UR}!n_{LL}!n_{LR}!} \quad (8)$$

Where:
(n):number of object.
(n_{UL}):number of object on the upper-left.
(n_{UR}):number of object on the upper-right.
(n_{LL}): number of object on the lower-left.
(n_{LR}):number of object on the lower-right.

In this work, we consider eight types of GUI defects that are detected using the structural metrics when their values go beyond specific thresholds. The aim is to compare these measures with an adequate threshold value. In Fig. 1, we present an illustrative example portraying the incorrect layout defect of the home interface of Reddit app: (left) Incorrect layout and (right) correct layout. This evaluation is based on the regularity metric which has a value more than the recommended threshold value which leads to incorrect layout defect.

This paper presents an empirical study that quantifies the impact of GMUI defects on XML files maintenance of Android apps in means of the change frequency with which developers maintain the infected files before and after their infection. The following section details the design of our empirical study.

4 Empirical Study Definition and Design

4.1 Illustrative example

Our work aims to investigate the effect of the aesthetic defects on the change-proneness of XML files related to a given mobile user interface. During the development of mobile apps, developers update them by adding a new feature, fixing errors, etc. Among those ongoing changes, developers/designers may partially or fully modify the design of the UIs from one update to another. To address negative end-user feedback, some design improvements of user interface can be introduced. In this study, we focus on determining the size of code modifications that developers/designers need to commit to correct one or more design problems. To better understand the motivation behind this work, we evaluate ten different versions of Home UI taken from ten different releases (from V1.2.1 to V1.2.3) of Reddit application. Figure 2 shows an illustrative example portraying the diffuseness of GMUI defects in Home UI of Reddit app. The depicted interface in the left of the first version (V 1.2.1) has two GMUI defects (Overloaded MUI and Incohesion MUI). The screenshot shown in the right is an intermediate version (V 1.2.1.5) which has six GMUI defects (overloaded MUI, Incohesion of MUI, Imbalanced MUI, Difficult navigation, Incorrect data presentation and Complicated MUI). As shown in Fig. 3, we tracked the ratio of commits that have incorporated code changes related to this interface. In fact, we note the existence of an important amount of rework (change size= 125), in the version (V

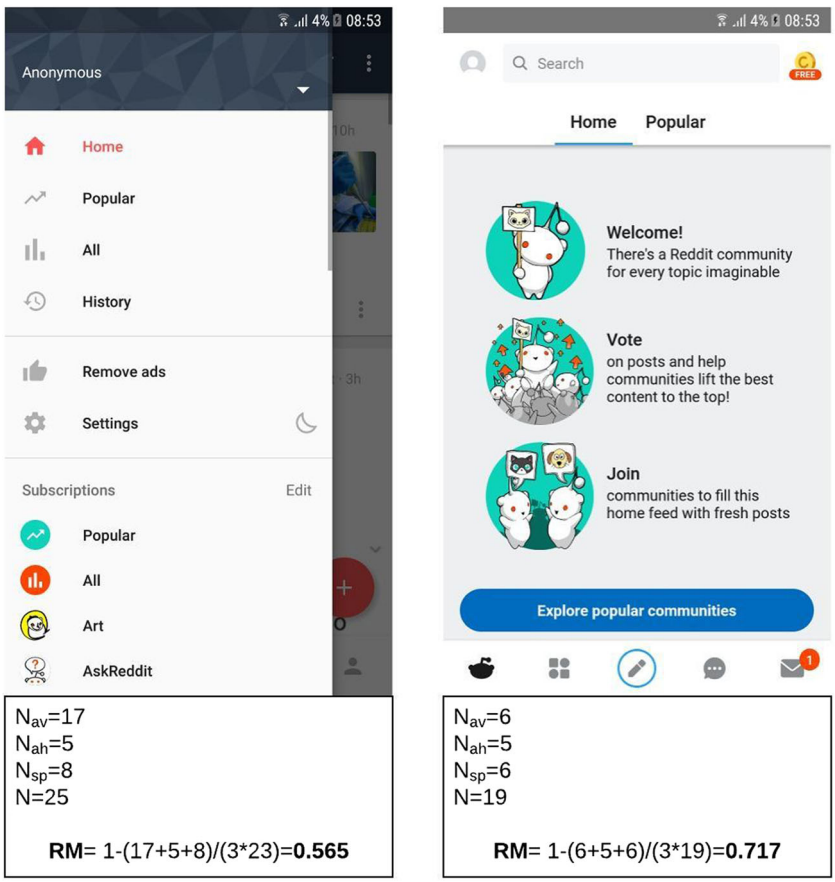

Fig. 1 An example of Regularity metric calculation of "Home" UI of "Reddit" app from two different releases

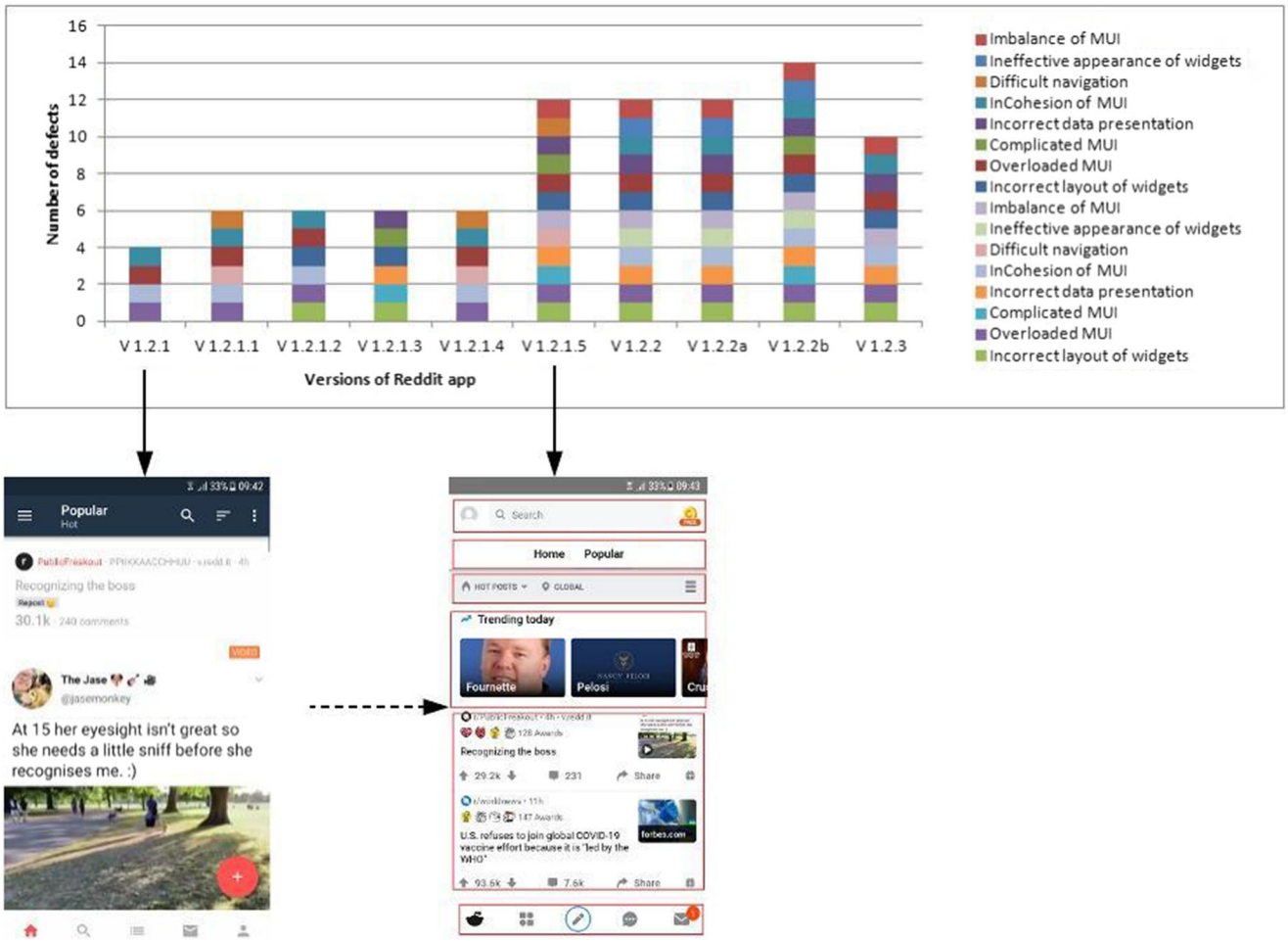

Fig. 2 An example of the "Home" UI of the "Reddit" app

1.2.1.5), when compared to the rework in the first version (change size= 0). Moreover, we mention the increase in the density of these changes right after the scaling up of the aesthetic defects. Therefore, we investigate whether the diffuseness of GMUI defects is heavily correlated with the increase in change size and frequency. Since it is difficult to confirm whether the existence of defects is responsible for such observation, we decided to conduct an empirical validation that we present in the following subsection.

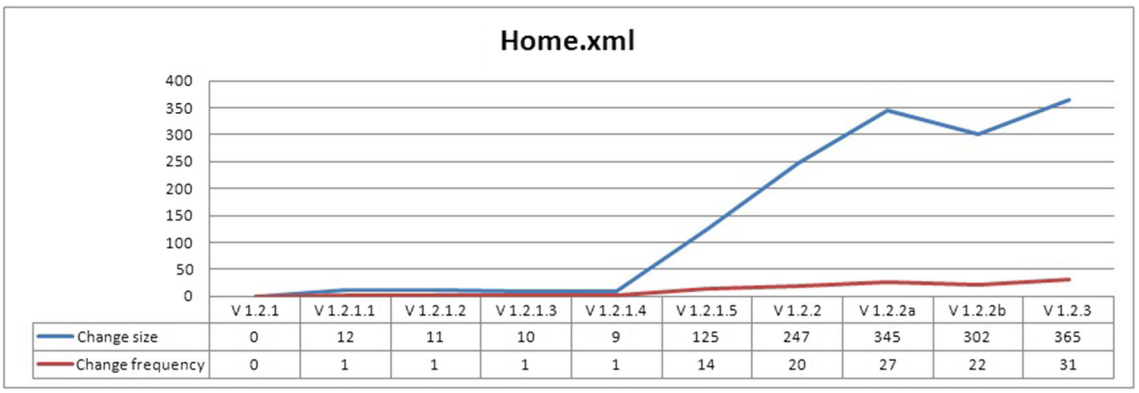

Fig. 3 An example of the "Home" UI of the "Reddit" app showing how change and frequency size rates increase after the introduction of aesthetics defects

4.2 Empirical Study Design

As shown in Fig. 4, our approach consists of producing three insights: 1) we investigate whether infected GMUI classes are changed more frequently and proportionally along with the system's evolution, and have a more substantial change-size than non-infected GMUI classes. 2) we study whether particular kinds of aesthetics defects are persistent than others. 3) we discover whether there is a co-occurrence between the studied GMUI defects.

The idea consists in controlling the change frequency and change-size of GMUI classes. We sought precisely to study the diffuseness and the impact of aesthetics defects on change proneness of the infected XML classes. Moreover, we analyze how specific GMUI defect could lead to different change size. Our aim regards the totality of XML files for each GMUI of a release.

The detailed steps of our empirical study are as follows: 1) We use PLAIN plugin to detect eight types of common GMUI defects across a total of 54 releases of five popular and open source mobile apps. 2) We categorize the GMUI classes into infected and non-infected ones. 3) We track the changes in all commit between releases from the GitHub platform. 4) We measure the LOC change (LOCC) per class for each version of the same app as being an essential parameter for the change-size calculation. 5) We calculate the CF and the CS for each class. Finally, 6) we study data dispersion of infected and non-infected classes. i.e., we analyze the survivability of GMUI defects and their co-occurrences.

4.3 The Studied Projects

Our work aimed at understanding the diffuseness of GUI defects in 5 selected Android applications from the Google Play Store. All the experimented data are available (Chouchane). **Mattermost:** is a secure messaging app that connects to servers from behind your firewall. It is used by thousands of companies around the world in 14 languages, and runs on Android and iOS. We analyzed 11 releases of Mattermost in the years 2016-2017 from release V-1.1.4 to V-3.10. In 2018, the application was extended to include 17 versions which indicate its popularity. **Openlauncher:** a native full open source Android launcher application. It supports many features as Double tap to sleep, Item customization on the desktop, and so many others. We analyzed 12 releases of Open launcher available on GitHub in 2017. It has a rating of 4.1 on Google Play store. **Weather:** a very popular kind of applications, that forecasts weather conditions in your city and the globe. We analyzed 9 versions between 2016-2017. It has 4.6 rating on Google Play store. **Reddit:** it provides users with all top trending topics, breaking news, viral video clips, and so on. We analyzed 11 releases of Reddit between 2013 and 2017. It has an average rating of 4.6 in the Google Play store. **Lightning:** a lightweight fast web browser that uses simple material design and gives the users lots of options to protect their privacy. This application is very popular on Google Play store as it has a paid version. We analyzed 12 releases from 2015 to 2017. It has a rating of 4.1 on Google Play store. Table 2 presents the applications technical characteristics.

The number of selected projects relates to the previous studies that conduct any manual and qualitative analysis. We verified that these apps represent a good sample by testing whether they satisfy the constraints of a well-engineered project (Munaiah et al. 2017). We also made sure that they are open source since our experiments rely on the analysis of the code base of these apps along with all their commits, to replicate their evolution over releases.

4.4 Research Questions

Based on the extracted data from each application, we are looking to answer the following questions. The corresponding hypotheses for this study are formulated one-sided.

To what extent are the considered GMUI defects diffused in the studied apps?

We aim to track the diffuseness of GMUI defects in the studied mobile apps to discover the highly diffused GMUI defects and hence prioritize their corrections.

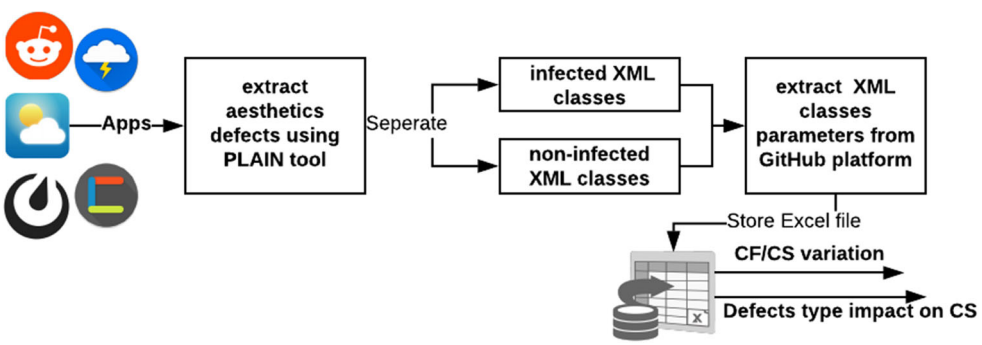

Fig. 4 Overview of our empirical study

Table 2 Characteristics of the tested Android applications

Char	Lightning	Mattermost	Weather	Openlauncher	Reddit
#Versions	12	11	10	12	11
#Revisions	1656	744	3140	2119	5374
Time-frame	2015-2017	2016-2017	2016-2017	2017	2013-2017

Char*= Characteristics

RQ2 *Is the application more susceptible to GMUI defects through its several updates?*

In this research question, we used PLAIN plugin in order to extract the GMUI defects in each release. The aim of this RQ is to observe the existence of aesthetics defects along with the releases by testing the following hypothesis: $H2_0$: The application is more susceptible to GMUI defects through its evolution.

RQ3 *Are infected GMUI classes more change-prone than non-infected GMUI classes?*

In RQ3, we are interested to see whether developers mostly focus on refactoring infected classes over time by testing the following null hypotheses:

$H3_0$: The Change Frequency (CF) of infected GMUI classes is equal to the CF of non-infected GMUI classes.

In this context, the change frequency (CF) refers to whether a class underwent at least a change between version v (participating or not in a defect) and the subsequent version (v+1). The CF is measured as the number of commits for each class.

$$CF(C_t) = \frac{\sum_{c=1}^{n} NC(C_t) * 100}{LOCC(C_t)} \tag{9}$$

Where: C_T: class C at time t; NC(C_t): returns the number of changes made in class c between revision n and revision n-1. LOCC(C_t): returns the Lines Of Change (LOCC) per class for each version of the same app.

RQ4 *Do infected GMUI classes require more LOC change than non-infected classes?*

In RQ4, we examine whether the presence of aesthetics defects leads to a significant increase in the number of touched lines of code by testing the following null hypothesis: $H4_0$: The CS of infected GMUI classes is equal to the CS of non-infected GMUI classes.

In this context, we have relied on the calculation of the change-size (CS) in which we measure the number of code lines that have been changed within a class in a release. i.e., (addition/removal/modification).

$$CS(C_t) = \frac{\sum_{c=1}^{n} CSIZE(C_t) * 100}{LOCC(C_t)} \tag{10}$$

Where: CSIZE(C_t): returns the sum of code changes on class c between revision n and revision n-1; LOCC(C_t): returns the LOC change (LOCC) per class for each version of the same app.

To test our hypotheses, the change frequency (CF), and the change-size (CS) were calculated for the infected and non-infected GMUI classes. Values are multiplied by 100 to avoid problems with rounding numbers when calculating CF and CS. A nonparametric Mann-Whitney U-test is opted similarly to Olbrich et al. (2010) since the data are abnormally distributed and the sample size is small (Sheskin 2003). An alpha value of 0.1 was used to deal accurately with our observations that do not exceed 40. For the sake of visibility, and since there are no substantial variations of the number of classes across releases, we aggregated data obtained from the releases of each application, rather than for each release separately.

What are the aesthetic defects of GMUI that persist throughout the releases of studied apps?

In RQ5, we predict the survivability of GMUI defects based on the occurrence of each aesthetic defect per release. We calculate the total number of user interfaces and the total number of GUI defects of each release. Then, we observe the evolution of each defects along the GMUIs evolution in order to discover the persistent aesthetic defects.

To what extent do GMUI defects co-occur?

In RQ6, we investigated how often the presence of GUI defects, per type (e.g., overloaded GUI) in the mobile user interface leads to the presence of another type of GUI defects (e.g., difficult navigation). To this end, we computed the number of defects in each GMUI and the percentage of GMUI affected by one or more types of aesthetic defects. Then, for each GUI Defect type (GD_i), we computed the percentage of its co-occurrence with another GUI Defect type (GD_j) in a given GMUI using the following formula (Palomba et al. 2018):

$$Co - occurrence_{i,j} = \frac{GD_i \cap GD_j}{GD_i} \tag{11}$$

Where i $\neq$ j. $GD_i \cap GD_j$ is the number of co-occurrences of defect i and defect j. GD_i is the number of occurrences of defect i.

4.5 Variables Selection

To answer our null hypotheses, we construct the analysis models based on the specification of the following dependent and independent variables related to each research question.

Dependent variables RQ3 and RQ4, we use the Mann-Whitney U-test to understand whether the change frequency (RQ3), and the change-size (RQ4) differ based on class type. i.e., our dependent variables would be the ”CF” for RQ3 and ”CS” for RQ4

Independent variables RQ3 and RQ4, would be the ”class type”, which has 2 groups (infected GMUI and non-infected GMUI).

In RQ5, we extract the existence of 8 types of GMUI defects. Each variable $G_{c,d,v}$ refers to how many instances of a defect d a GMUI class c has in a version v.

5 Analysis of Studied Results

Results of RQ1 To study the diffuseness of GMUI defects, we aggregate its occurrence number in all the releases of the five applications. The box plot brings out significant differences in the diffuseness of aesthetics defects. We use the box plot of the number of aesthetics defects instances in our analyzed Android apps.

From the box plot shown in Fig. 5, we note that overloaded (OM) defect is the highly frequent and diffused defect in all the apps followed by Complicated MUI and Difficult navigation defects. However, the other defects are neglected compared to the three mentioned defects. In addition, the detection of these defects by PLAIN helps developers in the maintenance process of these applications.

Results of RQ2 We calculate the percentage of GMUI aesthetic defects along with the application evolution. In

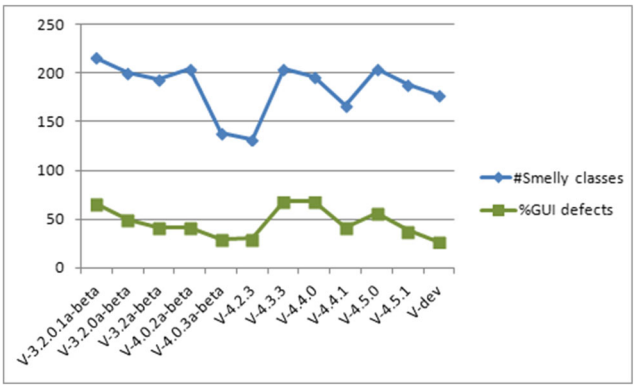

Fig. 6 Defects density of Lightning

Figs. 6, 7, 8, 9 and 10, the x-axis represents the studied releases of each app, the y-axis on the left side represents the total number of aesthetic defects, and infected GMUI classes per release.

$H2_0$: *The application is more susceptible to GMUI defects through its evolution.*

The analysis of GMUI defects occurrence frequency through the five apps evolution, denotes that the applications did not have more smells over updates. Referring to graphs 6 to 10, we can conclude that this statement is totally dependent on the number of infected GMUI classes in each release. It is noticeable that for the five applications, the behavior of GMUI aesthetics defects curve did follow the rate of the infected classes curve. So, we fail to accept ($H2_0$).

Results of RQ3, RQ4 Tables 3, 4, 5, 6 and , 7 show the results of the hypotheses tests. In all the tables, a gray-shaded p-value row indicates that the null hypothesis is rejected and thus the alternative hypothesis is supported. For all the tests, we used a one-sided test because we investigate only whether GMUI defects relate to an increase in the change frequency and change-size. In the result tables presented below, n represents the sample size, M for the median, sd

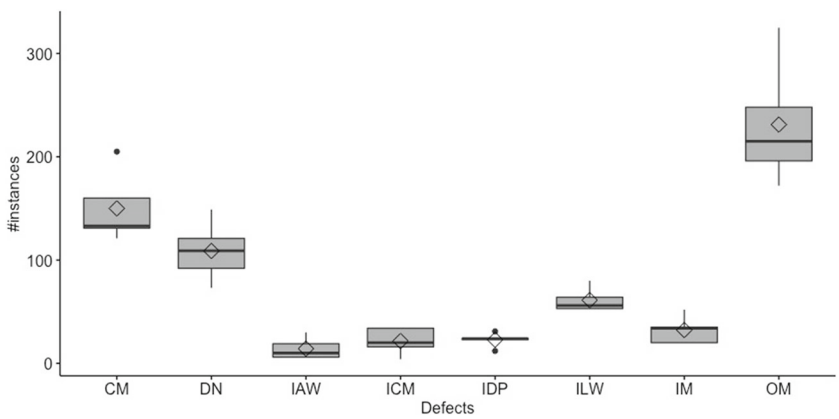

Fig. 5 Number of Aesthetics defects instances in the analyzed applications

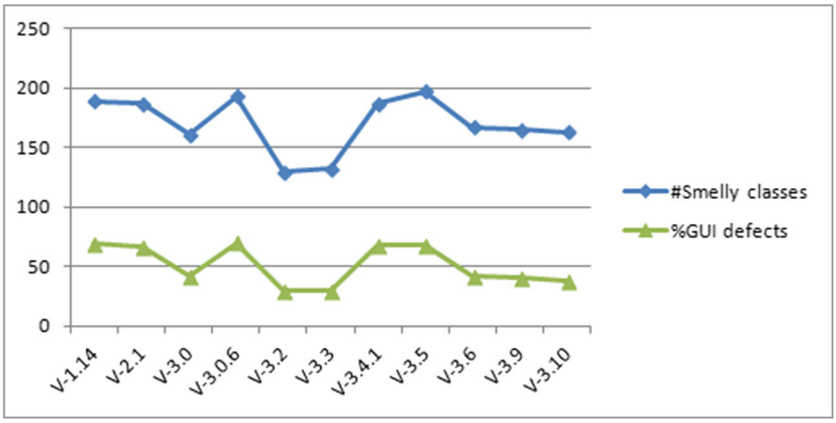

Fig. 7 Defects density of Mattermost

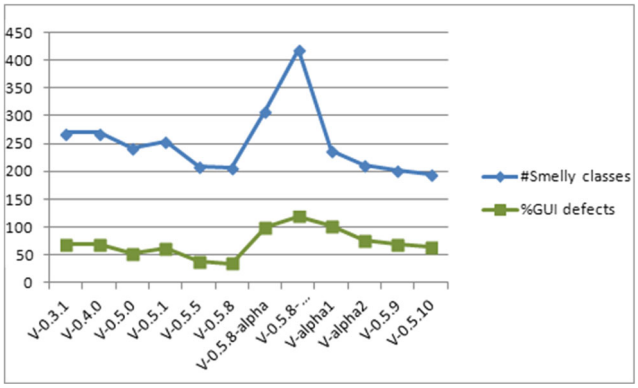

Fig. 8 Defects density of Openlauncher

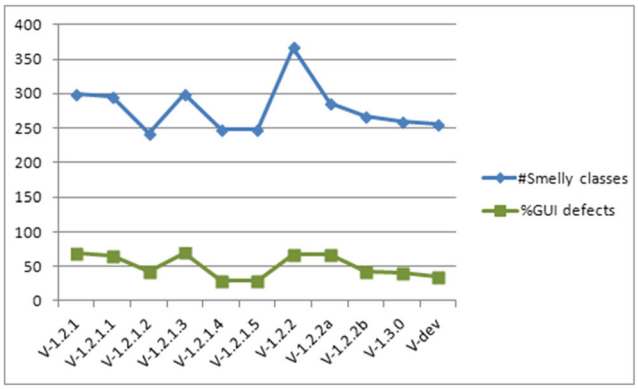

Fig. 9 Defects density of Reddit

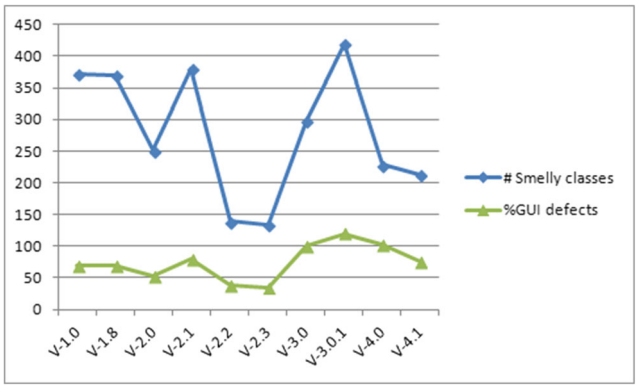

Fig. 10 Defects density of Weather

Table 3 Results for lightning app

		Infected classes	Non-Infected classes
	n:	15	11
CF	M:	0.523	0.228
	sd:	0.238	0.202
	P:	0.0005338	
	U:	146	
CS	M:	340	96
	sd:	206	92.9
	P:	0.0001141	
	U:	154	

Table 4 Results for Reddit app

		Infected classes	Non-Infected classes
	n:	21	10
CF	M:	0.489	0.0195
	sd:	0.144	0.0904
	P:	6.096e-06	
	U:	209	
CS	M:	496	13.5
	sd:	130	5.86
	P:	5.026e-06	
	U:	210	

Table 5 Results for weather app

		Infected classes	Non-Infected classes
	n:	7	6
CF	M:	0.799	0.0256
	sd:	0.138	0.0425
	P:	0.001703	
	U:	42	
CS	M:	879	23
	sd:	161	9.01
	P:	0.001703	
	U:	42	

Table 6 Results for Mattermost app

		Infected classes	Non-Infected classes
	n:	7	6
CF	M:	0.799	0.0256
	sd:	0.138	0.0425
	P:	0.001703	
	U:	42	
CS	M:	699	10
	sd:	108	6.95
	P:	0.001681	
	U:	42	

for the standard deviation, U for the calculated U-test, and P for P-value.

$H3_0$: The CF of infected GMUI classes is equal to the CF of non-infected GMUI classes.

The hypothesis is rejected by all five apps. Tables 4, 5, 6, 7 and 8 show that the medians of CF for infected GMUI Classes are remarkably (3-25-30-30-27 times) higher than for non-infected GMUI classes. These results show that each infected GMUI class line of code (LOC) is changed more often than non-infected classes.

$H4_0$: The CS of infected GMUI classes is equal to the CS of non-infected GMUI classes.

Tables 4, 5, 6, 7 and 8 show that the size of changes performed per line of code in the infected GMUI classes is, noticeably, larger than for non-infected classes. For example, in Mattermost app, the change-size for infected GMUI classes is, on average, 60 times higher than non-infected GMUI classes. The alternative hypothesis is supported by all five applications.

Table 7 Results for openlauncher app

		Infected classes	Non-Infected classes
	n:	42	11
CF	M:	0.683	0.0567
	sd:	0.157	0.116
	P:	2.148e07	
	U:	462	
CS	M:	892	209
	sd:	460	93.2
	P:	2.149e-07	
	U:	462	

Table 8 Co-occurrence GUI defects of studied mobile apps

	ILW	OM	CM	IDP	ICM	DN	IAW	IM
ILW	–	2%	12%	17%	0%	0%	2%	45%
OM	2%	–	**35%**	10%	7%	10%	8%	3%
CM	12%	**35%**	–	0%	0%	10%	5%	0%
IDP	17%	7%	0%	–	2%	0%	12%	0%
ICM	0%	7%	0%	2%	–	0%	0%	0%
DN	0%	10%	10%	0%	0%	–	4%	25%
IAW	2%	8%	5%	12%	0%	4%	–	12%
IM	**45%**	8%	0%	0%	0%	25%	12%	–

Results of RQ5 To track the evolution of each user interface, we compute the number of defects in each release of the studied mobile apps. From the figures (Figs. 11, 12, 13, 14 and 15), there are slightly frequent defects in the applications such as Incorrect appearance of widgets, InCohesion of MUI, Incorrect data presentation, and Imbalanced MUI. For instance, we found that the highest number of Imbalanced MUI instances is 11 in the V3.2.0a-beta of Lightning app, leading to 0.2 probability of a class getting infected by this defect. However, it exists in 69.6% of the releases. Incorrect appearance of widgets is also weakly diffused. It affects 41% of the releases and in the most two affected releases (weather V3.0.1, and OpenLauncher V5.8 (alpha)), only 21%, and 14.8% respectively are instances of this defect. The InCohesion of MUI affects 48.2% of the releases, with a probability of 0.26 of classes to get infected by this defect type. The highest number of instances of this defect in a single release (OpenLauncher V alpha2) is 15. In particular, the classes affected by the InCohesion of MUI in OpenLauncher are 65 out of 101 (65%). Other aesthetics defects are in opposite quite diffused. For example, we found that the Overloaded MUI defect is present in 100% of the analyzed releases, with a probability of 71% of appearing in a class. In particular, weather V3.0.1 (V8 in the graph in Fig. 15) has the highest number of this defect type (40 instances) in a total number of 57 classes since it is affecting 70.17% of the classes. The average of the 56 releases is affected by 20 Overloaded MUI, with 27 in OpenLauncher. Moreover, the Complicated MUI defect exists in the totality of the releases, with a probability of 0.46 of being present in a class, with the highest number of instances (26) found in a weather release V3.0.1. We conclude that Overloaded MUI (OM) and Complicated MUI (CM) are the top two GMUI defects that tend to survive in the studied releases with a probability of 100% followed by the Difficult navigation with 96.42% of affected releases, leading to 0.41 probability of a class getting infected by this defect.

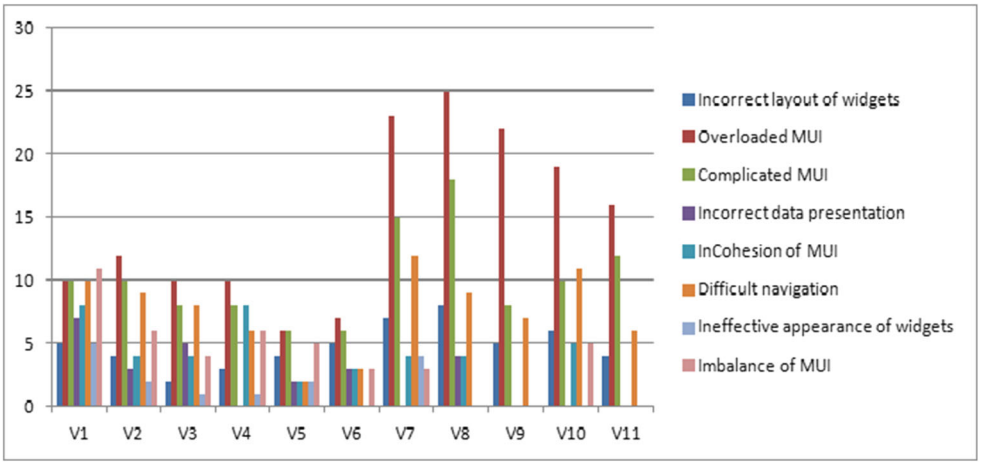

Fig. 11 Defects distribution of Lightning

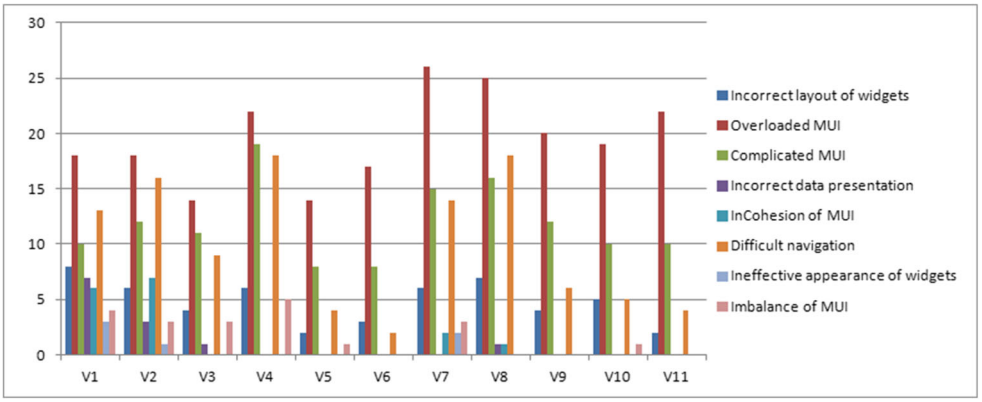

Fig. 12 Defects distribution of Mattermost

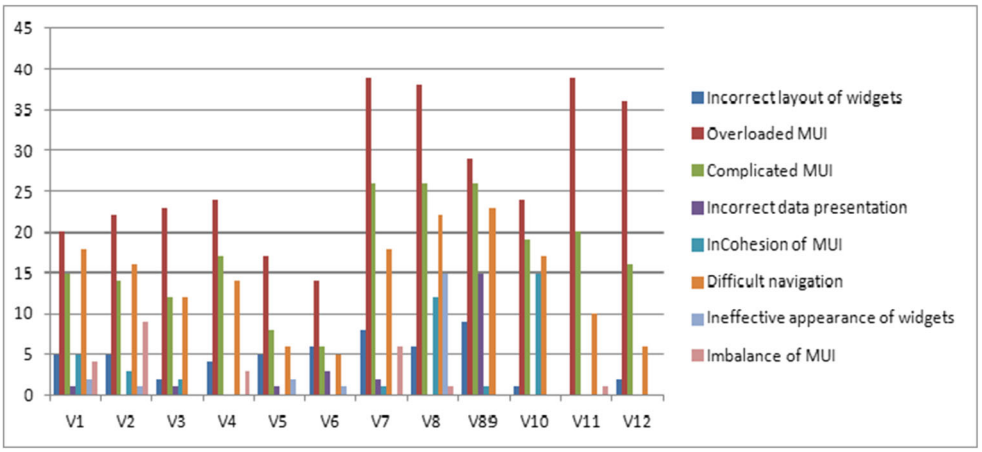

Fig. 13 Defects distribution of Openlauncher

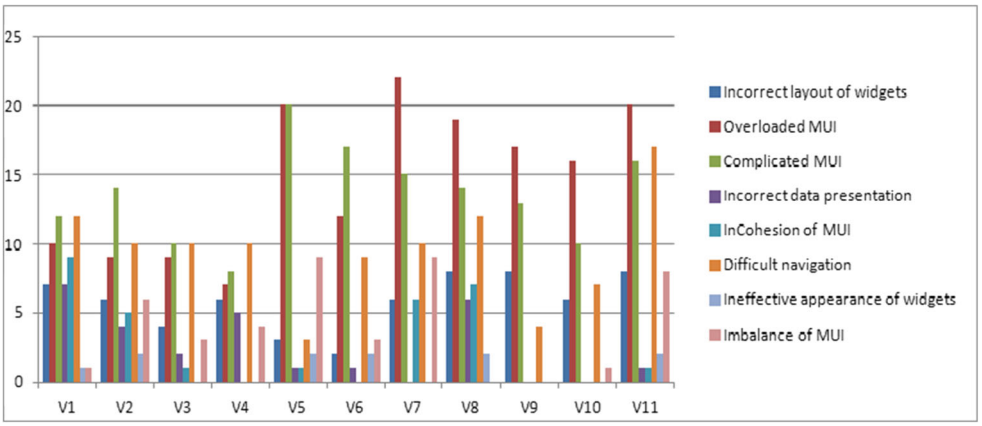

Fig. 14 Defects distribution of Reddit

Interestingly, we choose to show the evolution of one UI (Weather overview UI) for the sake of clarity as seen in Fig. 16. We evaluated these two UIs by PLAIN and we detected the following defects respectively OM, CM, IM, IAW, ILW, DN, IDP, and OM, IAW, ILW.

The first thing that leaps to the eyes, is the density of defects the weather overview UI has in V1.1 comparing to the weather overview UI in V2.0. The number of defects dropped down from 7 to 3. Weather overview V1.1, has a relatively large quantity of elements making it look charged. Although, the UI V2.0 has a minimum and structured elements, the non-inter-relatedness of layouts makes it feel overloaded. While some new design materials, and more structuring of the widgets layouts have been added to V2.0, the Overloaded MUI and the Incorrect layout of widgets survive. It means that the OM and CM are the most persistent defects along the interface evolution.

Results of RQ6 we study the co-occurrence between the GMUI defects in order to detect any potential dependencies. To this end, we computed the number of occurrences of each GUI defect in a given mobile app. Then, we determine the co-existence of different types of defects such as ILW and IM. The finding results, shown in Table 9, highlight the phenomenon of GUI defects co-occurrence for each pair of GUI defect types GD_i and GD_j.

As shown in Table 8, we observe that there exist two pairs of GUI defects types frequently co-occurring such as (ILW, IM) and (OM, CM). In the other side, we observe that there are other pairs including GUI defects slightly co-occur such as (DN, IDP) and (DN, IM). Among these, the Overloaded MUI defect (OM) is the defect that tend to co-occur more with other types, and in particular with Complicated MUI (CM) (i.e., 35% of GMUIs affected by OM are also affected by CM), IDP and DN (10%), IAW (8%) and ICM (7%). These findings can help developers to fix some defects type at the same time.

6 Discussion

In this section, we discuss the results of our experiments, and we provide the implications of the study for research and software engineering community.

6.1 Experiment Results Analysis

RQ1 and RQ2 In RQ1, we study the diffuseness of GMUI defects in the studied mobile apps. As shown in Fig. 5, we

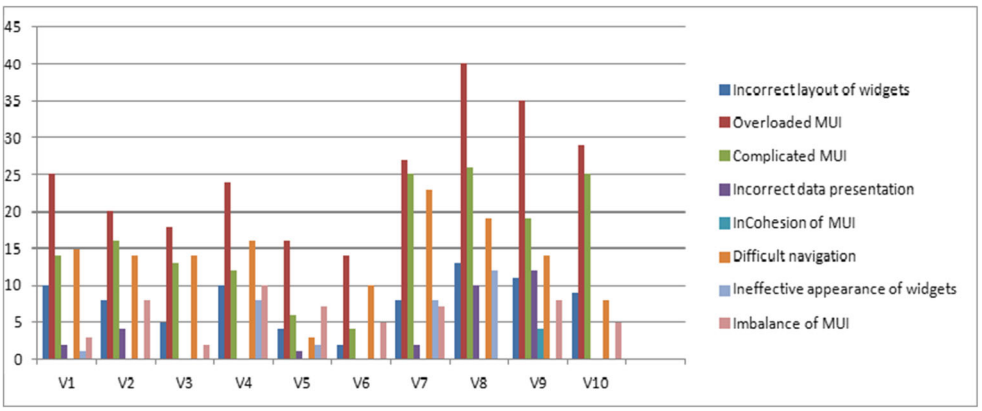

Fig. 15 Defects distribution of Weather

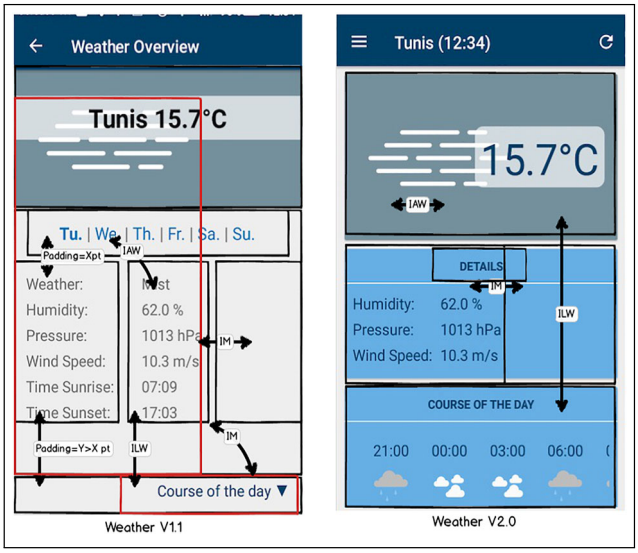

Fig. 16 Weather Overview UI in two consecutive releases V1.1 (left), and V2.0 (right)

can conclude that there are three GMUI defects which are the most frequent and diffused ones (OM, CM and DN) in all the releases of studied apps. On the contrary, (IDP, ILW and IAW) are poorly diffused. Moreover, referring to the Figs. 6-10, we can deduce that the curves of infected GMUI classes and aesthetics defects evolve at the same pace. At this stage of the study, the intensity of the correlation can be justified based on the behavior of bad defects in relation to the number of infected classes (GMUI), and it is interesting for our future work to know precisely the root causes behind this correlation. However, we can notice that in the case when the change-size decreases from a release to the subsequent, the number of defects decreases as well and Vice Versa. Thus, when developers make lots of changes to a class, there is a high chance of jeopardizing the quality of the UI and produce many additional defects. This interpretation raises the possibility of other external factors interfering in such conduct.

RQ3 and RQ4 From Tables 4, 5, 6, 7 and 8, it can be noticed that infected and non-infected GMUI classes were significantly different regarding their change frequency and change-size. We showed that CS and CF of infected classes are significantly larger than respectively the CS and CF of non-infected classes. This result is not surprising since maintenance activities will target mostly infected classes to clean out the defects. We can conclude that the presence of aesthetics defect has an impact on the change-proneness and change-size of a class.

RQ5 and RQ6 Based on Figs. 11-15 and Table 9, we conclude that there are types of defects which are more persistent compared to other defects. However, during the evolution of the five studied mobile apps, we discover that overloaded MUI is the defect that has the highest number of occurrences in all the releases followed by Complicated MUI and Difficult Navigation defects. Moreover, we observe that there are two pairs of GMUI defects that are present simultaneously in the studied GMUIs. These pairs are (ILW, IM) and (OM, CM). Furthermore, these findings can help developers in their maintenance tasks because detecting the persistence of each defect can help them to prioritize their improvement changes and try to fix the highly frequent GMUI defects. In addition, discovering the defects type frequently co-occurring in the same release may help developers to fix them at the same time.

6.2 Implications for Research

Based on recent studies on GMUI evaluation (event-driven level and structural level), developers are quite enforced to perform separate maintenance operations on both java source code and XML files. This practice will result in maintenance workload and time consumption whenever functional and structural GMUI defects are detected. Taking into account the survivability of defects types provides guidance to software engineers on improving their maintenance operations. Furthermore, It would be interesting to investigate the survivability of these defects types on two levels: 1) the survivability of the defects on a class change-proneness.i.e., how much LOC changes a defect type requires to fix it? 2) the survivability of the user's satisfaction. From a user point of view, which defect is more likely to deteriorate the usability level of the interface? These hypotheses can be further investigated in future studies.

Our results can be of interest to developers, who need to know the impact of defects on their maintenance activities, in order to predict their effort and workload. Knowing the effect of an existent defect will help developers to choose between two scenarios: 1) maintain some GMUI design problems to do the minimum modifications. This solution can be of interest to managers when there is a time-line that must be respected. 2) choose to fix the most effort requiring defect to provide better usability or UX.

7 Threats to Validity

In this section, we present factors that may impact the applicability of our observations in real-life situations.

7.1 Internal Validity

It raises potential concerns regarding any factors that may attenuate the observations. For our work we rely on change-frequency and change-size as two main measurements we

are correlating with the existence of defects. In fact, many factors may also be responsible for increasing the proneness and size of changes. For instance, the change frequency may be easily be implied by the importance of the UI. If an interface represents the home-screen of the app, then, eventually, it would be undergoing an important set of regular updates. This is mitigated by investigating several releases of the same interfaces as the change frequency across releases will not impact our model unless there is a drastic change that may be captured in a major release.

Another relevant example, which may trigger a wider set of changes, is the density of the interface, i.e., UIs with more services tend to contain a higher number of components to maintain. To mitigate this, our model normalizes over the LOC. Furthermore, the density of an interface of also considered as part of the defects and it would be relevant for our study to compare interfaces with various number of components. The data was collected from the applications hosted repositories on GitHub. Based on each contributor's changes, we were able to get the related change frequency and the change-size for each class of the application. However, we didn't consider the quality of contributors, that we claim to be a powerful effect factor on our experimental results.

7.2 Construct Validity

It concerns the tools used in our data collection and analysis. In our context, we used PLAIN to detect defects. Although the precision of PLAIN has been previously assessed (Soui et al. 2017), any false positives issued by the tool has a direct impact on our study.

Furthermore, we manually verified the extraction of the information used for the experiments since the number of releases and projects is reasonable. However, manual activities can relatively increase the error rate that might infect our measurements.

7.3 External Validity

We have purposely chosen 5 different Android apps to diversify the UIs under analysis. We also explored the evolution of UIs by visiting various releases of each app. This diverse set of UIs containing different structures and functionalities that strengthen the generalization of our observations. Yet, we would like to extend our dataset and perform a larger-scale empirical study to challenge our current findings.

8 Conclusion

In this paper, we investigated an empirical study, performed on five applications, each of which has more than nine releases. It provides a clue that aesthetics defects presence do influence a class change frequency and change-size. We assessed the relationship between the CS and CF of infected and non-infected GMUI classes, which resulted in a robust significant difference for the five applications. Moreover, we investigate the survivability of GMUI defects as well as the co-occurrence between these defects in order to prioritize their correction.

In fact, some designers might choose to fix the defect with the minimum impact on change-size. However, others might prioritize the visual aesthetic attractiveness of the user interface, and choose to fix the riskiest defect. In this context, we plan to study the impact of change-size on the visual attractiveness of GMUI. In addition, we plan to design new GMUI defect detectors able to suggest appropriate refactoring operations in presence of GMUI defects co-occurrences. To generalize our findings, we plan to extend the number of experimented applications in order to validate our results by considering more programming context. It would be interesting also, to take into account the number of collaborators and their quality, which we assume to be an indispensable agent in the maintenance activities. This insight will let us know how to measure the risk of having a defect while modifying an application source code. Furthermore, we plan to study the impact of removing or introducing a persistent defect on the change-size and change proneness. Another direction worth to explore is to enhance the aesthetic quality through a set of refactoring operations of mobile user interface.

References

Akiki, P. A., Bandara, A. K., Yu, Y. (2014). Adaptive model-driven user interface development systems. *ACM Computing Surveys (CSUR)*, *47*(1), 9.

Alemerien, K., & Magel, K. (2014). Guievaluator: A metric-tool for evaluating the complexity of graphical user interfaces. In *SEKE* (pp. 13–18).

Alemerien, K., & Magel, K. (2015). Slc: a visual cohesion metric to predict the usability of graphical user interfaces. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (pp. 1526–1533): ACM.

AlOmar, E. A., Mkaouer, M. W., Ouni, A. (2019). Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *Proceedings of the 3nd International Workshop on Refactoring-accepted*: IEEE.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., Binkley, D. (2012). An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)* (pp. 56–65): IEEE.

Bavota, G., Qusef, A., Oliveto, R., Lucia, A., Binkley, D. (2015). Are test smells really harmful? an empirical study. *Empirical Softw. Engg.*, *20*(4), 1052–1094. https://doi.org/10.1007/s10664-014-9313-0.

Blouin, A., Lelli, V., Baudry, B., Coulon, F. (2017). User interface design smell: Automatic detection and refactoring of blob listeners. *Information and Sofwatre Technology*.

Chouchane, M. https://github.com/mabroukachouchane/correlation.

Constantine, L. L. (1996). Visual coherence and usability: a cohesion metric for assessing the quality of dialogue and screen designs. In *Proceedings Sixth Australian Conference on Computer-Human Interaction* (pp. 115–121): IEEE.

Fowler, M., & Beck, K. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Gao, J., Li, L., Bissyandé, T.F., Klein, J. (2019). On the evolution of mobile app complexity. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)* (pp. 200–209): IEEE.

González, S., Montero, F., González, P. (2012). Balores: a suite of principles and metrics for graphical user interface evaluation. In *Proceedings of the 13th International Conference on Interacción Persona-Ordenador* (p. 9): ACM.

Hartmann, J., Sutcliffe, A., Angeli, A. D. (2008). Towards a theory of user judgment of aesthetics and user interface quality. *ACM Transactions on Computer-Human Interaction (TOCHI)*, *15*(4), 15.

Ines, G., Makram, S., Mabrouka, C., Mourad, A. (2017). Evaluation of mobile interfaces as an optimization problem. *Procedia Computer Science*, *112*, 235–248.

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

Khomh, F., Di Penta, M., Guéhéneuc, Y. (2009). An Exploratory Study of the Impact of Code Smells on Software Change-proneness. École Polytechnique de Montréal, Tech. Rep. EPM-RT-2009-02.

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.

Li, W., & Shatnawi, R. (2007). An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, *80*(7), 1120–1128. https://doi.org/10.1016/j.jss.2006.10.018.

Masi, E., Cantone, G., Mastrofini, M., Calavaro, G., Subiaco, P. (2012). Mobile apps development: A framework for technology decision making. In *International Conference on Mobile Computing, Applications, and Services* (pp. 64–79): Springer.

Mercaldo, F., Di Sorbo, A., Visaggio, C. A., Cimitile, A., Martinelli, F. (2018). An exploratory study on the evolution of android malware quality. *Journal of Software: Evolution and Process*, *30*(11), e1978.

Mkaouer, M. W., Kessentini, M., Bechikh, S., Deb, K., Ó Cinnéide, M. (2014). High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using nsga-iii. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation* (pp. 1263–1270): ACM.

Mkaouer, W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K., Ouni, A. (2015). Many-objective software remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *24*(3), 17.

Mkaouer, M. W., Kessentini, M., Cinnéide, M.O., Hayashi, S., Deb, K. (2017). A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering*, *22*(2), 894–927.

Moorthy, A. K., & Bovik, A. C. (2011). Blind image quality assessment: From natural scene statistics to perceptual quality. *IEEE Transactions on Image Processing*, *20*(12), 3350–3364.

Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M. (2017). Curating github for engineered software projects. *Empirical Software Engineering*, *22*(6), 3219–3253.

Myers, A. C. (1995). Bidirectional object layout for separate compilation. In *ACM SIGPLAN Notices*, (Vol. 30 pp. 124–139): ACM.

Ngo, D.CL., Teo, L.S., Byrne, J.G. (2000). Formalising guidelines for the design of screen layouts. *Displays*, *21*(1), 3–15.

Norman, D. A. (2004). *Emotional design: Why we love (or hate) everyday things*. Basic Civitas Books.

O'Brien, H. L., & Toms, E. G. (2010). The development and evaluation of a survey to measure user engagement. *Journal of the Association for Information Science and Technology*, *61*(1), 50–69.

Olbrich, S. M., Cruzes, D. S., Sjøberg, D.I.K. (2010). Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *2010 IEEE International Conference on Software Maintenance (ICSM)* (pp. 1–10): IEEE.

Paiano, A., Lagioia, G., Cataldo, A. (2013). A critical analysis of the sustainability of mobile phone use. *Resources, Conservation and Recycling*, *73*, 162–171.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., De Lucia, A. (2017). Lightweight detection of android-specific code smells: The adoctor project. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 487–491): IEEE.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A. (2018). A large-scale empirical study on the lifecycle of code smell co-occurrences. *Information and Software Technology*, *99*, 1–10.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., De Lucia, A. (2019). On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology*, *105*, 43–55.

Parush, A., Nadir, R., Shtub, A. (1998). Evaluating the layout of graphical user interface screens: Validation of a numerical computerized model. *International Journal of Human-Computer Interaction*, *10*(4), 343–360.

research, A.BI. (2013). https://www.abiresearch.com/press/android-will-account-for-58-ofsmartphone-app-down.

Sears, A. (1993). Layout appropriateness: A metric for evaluating user interface widget layout. *IEEE Transactions on Software Engineering*, *19*(7), 707–719.

Sheskin, D. J. (2003). *Handbook of parametric and nonparametric statistical procedures*. CRC Press.

Shoaib, M., Shah, A., Majeed, F. (2011). Software design quality metrics for web based applications. *Pakistan Journal of Science, 63*(1).

Silvennoinen, J., Candidate, P., Vogel, M., Kujala, S. (2014a). Experiencing Visual Usability and Aesthetics in Two Mobile Application Contexts. *Journal of Usability Studies*, *10*(1), 46–62. http://www.upassoc.org.

Silvennoinen, J., Vogel, M., Kujala, S. (2014b). Experiencing visual usability and aesthetics in two mobile application contexts. *Journal of Usability Studies*, *10*(1), 46–62.

Soui, M., Chouchane, M., Gasmi, I., Mkaouer, M. W. (2017). Plain: Plugin for predicting the usability of mobile user interface. In *VISIGRAPP (1: GRAPP)* (pp. 127–136).

Soui, M., Chouchane, M., Mkaouer, M. W., Kessentini, M., Ghedira, K. (2019). Assessing the quality of mobile graphical user interfaces using multi-objective optimization. *Soft Computing*, 1–30.

Statista (2020). https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/.

Store, A. (2020). https://android.jlelse.eu/apple-vs-android-a-comparative-study-2017-c5799a0a1683.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D. (2016). An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016* (pp. 4–15). New York: ACM.

Türkyilmaz, A., Kantar, S., Bulak, M.E. (2015). User Experience Design: Aesthetics or Functionality? *Intellectual Capital and* . . . , 559–565. http://www.toknowpress.net/ISBN/978-961-6914-13-0/papers/ML15-111.pdf.

Yamashita, A., & Moonen, L. (2013). Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *2013 35th International Conference on Software Engineering (ICSE)* (pp. 682–691): IEEE.

Makram Soui received a PhD in Computer Science from the Polytechnic University of Hauts-de-France, in 2010. He holds many academic certificates such as IBM Predictive Analytics Modeler, IBM Business Intelligence Analyst, Oracle Certified Professional Java Programmer, IBM Cloud Application Developer, Microsoft 98-367 Security fundamentals. He is currently an Assistant Professor and in College of Computing and Informatics Saudi Electronic University, Riyadh. His responsibilities include: steering committee member for different branches of SEU across Saudi Arabia, head of research group "Business Intelligence", teaching various courses as the medium of instruction in English such as artificial intelligence, machine learning, datamining, to undergraduate and postgraduate students, revising and developing courses plans. He published 10 referred journal papers and 24 conference papers with a low acceptance rate (between 22% and 34%).

Mabrouka Chouchane is currently a PhD student in computer science at the University of Manouba, Tunisia. She received her master degree in 2015, from the higher institute of management of Gabes, tunisia. She works in the LARIA lab. Her main interests are in humancomputer interface design, mobile user interfaces and software engineering methods. She published 1 referred journal paper and 2 conference papers.

Narjes Bessghaier is a PhD at the Ecole de Technologie Supérieure (ETS) in Montreal (Canada). Her research interests include source code static analysis, mining repositories, software quality, code smells refactoring, as well as user interface evaluation. Her work has been published at major software engineering venues such as ACM Transaction on Interactive Intelligent Systems (ACM TIIS). She obtained her master degree in Enterprises Systems Engineering from the Institute of Computer science and Multimedia (ISIM) in Sfax-Tunisia.

Mohamed Wiem Mkaouer is currently an Assistant Professor in the Software Engineering Department, in the B. Thomas Golisano College of Computing and Information Sciences at the Rochester Institute of Technology. He received his PhD in 2016 from the University of Michigan-Dearborn under the supervision of Professor Marouane Kessentini. His research interests include software quality, systems refactoring, model-driven engineering and software testing. His current research focuses on the use computational search and evolutionary algorithms to address several software engineering problems such as software quality, software remodularization, software evolution and bug management.

Marouane Kessentini is a recipient of the prestigious 2018 President of Tunisia distinguished research award, the University distinguished teaching award, the University distinguished digital education award, the College of Engineering and Computer Science distinguished research award, 4 best paper awards, and his AI-based software refactoring invention, licensed and deployed by industrial partners, is selected as one of the Top 8 inventions at the University of Michigan for 2018 (including the three campuses), among over 500 inventions, by the UM Technology Transfer Office. He is currently a tenured associate professor and leading a research group on Software Engineering Intelligence. Prior to joining UM in 2013, He received his Ph.D. from the University of Montreal in Canada in 2012. He received several grants from both industry and federal agencies and published over 110 papers in top journals and conferences. He has several collaborations with industry on the use of computational search, machine learning and evolutionary algorithms to address software engineering and services computing problems.

Terms and Conditions

Springer Nature journal content, brought to you courtesy of Springer Nature Customer Service Center GmbH (“Springer Nature”).
Springer Nature supports a reasonable amount of sharing of research papers by authors, subscribers and authorised users (“Users”), for small-scale personal, non-commercial use provided that all copyright, trade and service marks and other proprietary notices are maintained. By accessing, sharing, receiving or otherwise using the Springer Nature journal content you agree to these terms of use (“Terms”). For these purposes, Springer Nature considers academic use (by researchers and students) to be non-commercial.
These Terms are supplementary and will apply in addition to any applicable website terms and conditions, a relevant site licence or a personal subscription. These Terms will prevail over any conflict or ambiguity with regards to the relevant terms, a site licence or a personal subscription (to the extent of the conflict or ambiguity only). For Creative Commons-licensed articles, the terms of the Creative Commons license used will apply.
We collect and use personal data to provide access to the Springer Nature journal content. We may also use these personal data internally within ResearchGate and Springer Nature and as agreed share it, in an anonymised way, for purposes of tracking, analysis and reporting. We will not otherwise disclose your personal data outside the ResearchGate or the Springer Nature group of companies unless we have your permission as detailed in the Privacy Policy.
While Users may use the Springer Nature journal content for small scale, personal non-commercial use, it is important to note that Users may not:

1. use such content for the purpose of providing other users with access on a regular or large scale basis or as a means to circumvent access control;
2. use such content where to do so would be considered a criminal or statutory offence in any jurisdiction, or gives rise to civil liability, or is otherwise unlawful;
3. falsely or misleadingly imply or suggest endorsement, approval , sponsorship, or association unless explicitly agreed to by Springer Nature in writing;
4. use bots or other automated methods to access the content or redirect messages
5. override any security feature or exclusionary protocol; or
6. share the content in order to create substitute for Springer Nature products or services or a systematic database of Springer Nature journal content.

In line with the restriction against commercial use, Springer Nature does not permit the creation of a product or service that creates revenue, royalties, rent or income from our content or its inclusion as part of a paid for service or for other commercial gain. Springer Nature journal content cannot be used for inter-library loans and librarians may not upload Springer Nature journal content on a large scale into their, or any other, institutional repository.
These terms of use are reviewed regularly and may be amended at any time. Springer Nature is not obligated to publish any information or content on this website and may remove it or features or functionality at our sole discretion, at any time with or without notice. Springer Nature may revoke this licence to you at any time and remove access to any copies of the Springer Nature journal content which have been saved.
To the fullest extent permitted by law, Springer Nature makes no warranties, representations or guarantees to Users, either express or implied with respect to the Springer nature journal content and all parties disclaim and waive any implied warranties or warranties imposed by law, including merchantability or fitness for any particular purpose.
Please note that these rights do not automatically extend to content, data or other material published by Springer Nature that may be licensed from third parties.
If you would like to use or distribute our Springer Nature journal content to a wider audience or on a regular basis or in any other manner not expressly permitted by these Terms, please contact Springer Nature at

onlineservice@springernature.com