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

Article *in* Information Systems Frontiers · April 2022

DOI: 10.1007/s10796-020-10100-w

CITATIONS
2

READS
68

5 authors, including:

Makram Soui
Saudi Electronic University
49 PUBLICATIONS **227** CITATIONS
SEE PROFILE

Mabrouka Chouchane
Ecole Nationale des Sciences de l'Informatique
7 PUBLICATIONS **40** CITATIONS
SEE PROFILE

Narjes Bessghaier
École de Technologie Supérieure
6 PUBLICATIONS **19** CITATIONS
SEE PROFILE

Mohamed Wiem Mkaouer
Rochester Institute of Technology
138 PUBLICATIONS **1,332** CITATIONS
SEE PROFILE

On the impact of aesthetic defects on the Maintainability of Mobile Graphical User Interfaces: An Empirical Study ☆

Makram Soui[1]

College of Computing and Informatics Saudi Electronic University, Saudi Arabia

Mabrouka Chouchane[2]

School of computer science of Manouba, Tunisia

Narjes Bessghaier[3]

School of computer science of Manouba, Tunisia

Mohamed Wiem Mkaouer[4]

Rochester Institute of Technology

Marouane Kessentini[5]

University of Michigan

Khaled Ghedira[6]

Honoris United Universities

Abstract

As the development of Android mobile applications continues to grow to follow up its high increase in demand and market share, there is a need for automating the evaluation of their graphical user interfaces and 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 interfaces by studying the connection between the existence of the defects and the change-proneness of interfaces. We empirically experiment the impact of 8 aesthetics defects in 56 releases of 5 Android applications, as we explore the following research questions: (1) are developers fixing the defects along the releases, (2) to what extent infected GUI classes are subject to change than other classes, (3) to what extent this subjectivity is due to the presence of defects. Our empirical validation confirms that (1) various defects typically exist in interfaces?, (2) the vulnerability of interfaces, infected with defects, to an increase of changes?, and (3) some aesthetics defects are more severe than others?.

Keywords: Aesthetics defects, change-size, Correlation, Evolution of Android GUI
2010 MSC: 00-01, 99-00

☆ Fully documented templates are available in the elsarticle package on CTAN.

[1] m.soui@seu.edu.sa
[2] chouchane.mabrouka@gmail.com
[3] bessghaier.narjess@gmail.com
[4] mwmvse@rit.edu
[5] marouane@umich.edu
[6] khaled.ghedira@universitecentrale.tn

1. Introduction

With the evolution of smartphones, mobile applications (apps) are becoming one of the pillars of software market URL (c). Nowadays, industry heavily relies on mobile apps to reach end-users swiftly and smoothly. Nearly 197 billion apps were downloaded in 2017 URL (d), and Android apps have been leading the market share with 87% in 2016 URL (a). 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 apps longevity 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 GUI level. Thereby, the good design of mobile Graphical User Interfaces (GUI) plays an important role in promoting the apps quality.

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 purpose. The need for a high visual attractiveness of the GUI 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 GUI, 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 GUI design aspect as it is mostly rated as low negative impact factor. However, GUI's code may achieve up to 45% of the total application Myers (1995), and though they cannot create 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 severity 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 along the apps evolution. We aim for 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 causing a larger chunk of source line-changes.

Our main findings prove that infected GUI source files experience an increase in the change-size in comparison with the noninfected ones. The change-size varies mainly depending on the type of the existent smells as it has been proven for Android code smells Khomh et al. (2012). We believe the same reason implies for the UIs, where we empirically shed the light to this matter. However, the approximation of defects severity is better performed using qualitative analysis and developers surveys. Therefore, investigating the severity of GUI defects is part of our future work. Furthermore, we notice a large variation on defects impact on files proneness to changes, which advocates prioritizing their correction with respect to severity.

The remainder of this paper is organized as follows: Section 2 defines the necessary terminology used in this paper. Section 3 presents our empirical study and main research questions while Section 4 provides the empirical validation. Section 5 discusses the key findings of the experiments while raising potential threats to their validity. The paper ends with Section 6 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 severity of internal and external code smells.
These studies Olbrich et al. (2010); Li and Shatnawi (2007); Khomh et al. (2009); Mkaouer et al. (2017) 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) has empirically investigated the correlation between two smells God and Brain classes with regard to their change frequency and change-size along 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 observations hold across all versions and projects. They also observe a variance on 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. These empirical studies highlight the importance for the community to develop tools to detect test smells and automatically refactor them. Thought the external defect are detected at the GUI level, they differ from the structural GUI defects that we are dealing with in this work. 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 interfaces. Pragmatically, are we deteriorating, maintaining or improving the UI structure as we modify the code? In fact, 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 GUI maintenance activities by studying the diffuseness of aesthetics defects for the same UI throughout several releases.

3. Background

3.1. GUI Aesthetic Defects

Samsung CEO Yun Jong Yong said- Good design is the most important way to differentiate ourselves from our competitors. 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 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. (2014). 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 aesthetics design cannot be ignored. It is the visual attraction between a user and an app. Donald Norman, the UX expert, sees beauty

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 of lack 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)

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. (2014) 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. Accordingly, aesthetics and performance should go hand in hand. Several tools have been proposed for MUI quality evaluation Soui et al. (2017),Bastien and Scapin (1995),Zen and Vanderdonckt (2014).

In this study, we used our tool PLAIN Soui et al. (2017), which detects a set of 8 aesthetic defects devoted to MUI evaluation. The detection is based on a set of 8 metrics inspired by Ngo et al. Ngo et al. (2000). We have adapted the desktop-oriented metrics to fit the mobile user interface context. PLAIN is based on the genetic algorithm technique used for the generation of evaluation rules Ines et al. (2017). This approach has reached 70% of precision and recall. Table 1, depicts the considered aesthetics defects and their definitions.

This paper presents an empirical study that quantifies the impact of GUI defects on XML files maintenance in 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 Design

4.1. Illustrative example

Users reviews may provide valuable information to help developers in improving an app's functionality or a GUI bug. This feedback is crucial to guide the developers app maintenance. A considerable amount of reviews are complaining about the usability of the user interface level. As this latter represents nearly 50 % of software code Myers (1995); Park et al. (2013), it is of importance to assess the required maintenance effort for different GUI structural issues. In Figure 1, we collected some users reviews complaining about how the UI of the Evernote application is confusing to the user. We provide as a sample the structural aesthetic defects of the notebooks GUI of the Evernote application. The mainly notable structural defect is the Imbalance of the GUI, that might be seen as an empty user interface (lack of features). An imbalanced GUI is aesthetically unpleasing and may provoke the need of employing

extra options. Improving the structural aesthetic feel and look of the GUI necessitates sometimes a drastic change of the GUI code. In this study, we assess eight structural problems and their impact on the change size of the XML code.

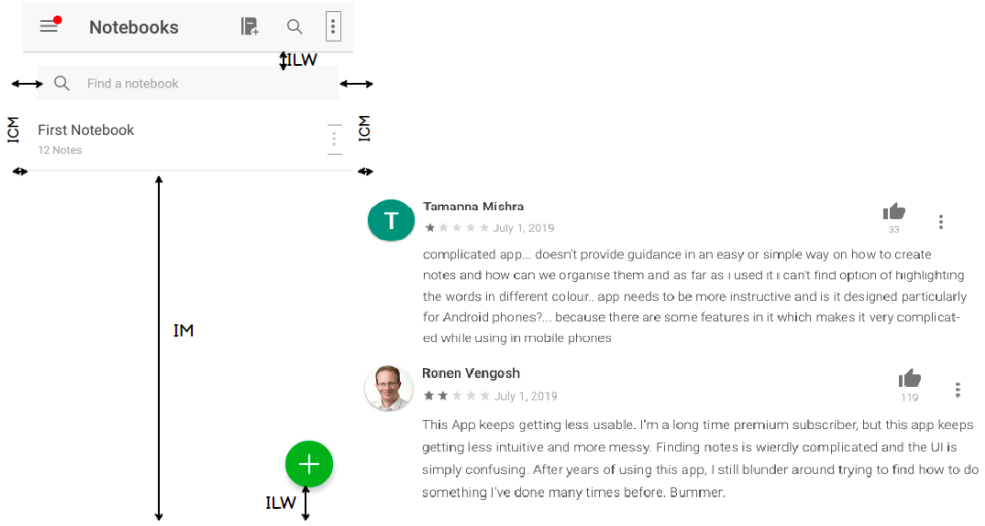

Figure 1: Complaints about the low usability quality of the Evernote app

4.2. Research Methodology.

The *goals* of our study are: 1) we investigate whether infected GUI classes are changed more frequently and proportionally along with the system's evolution, and have a more substantial change-size than non-infected GUI classes. 2) we study whether particular kinds of aesthetics defects necessitate more changes.

The idea consists in controlling the change frequency and change-size of GUI classes. We sought precisely to study the influence of infected GUI classes (XML files) change-size on the aesthetics defects. Our aim regards the totality of XML files for each GUI of a release. As shown in Figure 2, our approach consists of producing two insights.

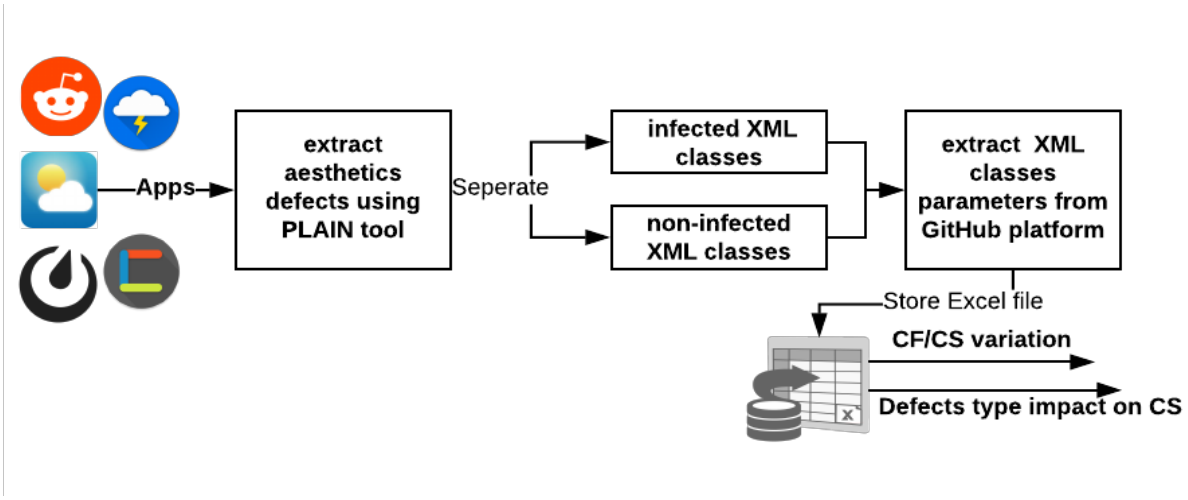

Figure 2: Research methodology.

4.2.1. CF, CS variation

The detailed steps of our empirical study are as follwos: 1) We evaluate aesthetics defects of the evaluated app. 2) We categorize the GUI classes into infected and non-infected ones. 3) We extract parameters from 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 whether the presence of GUI defects relates to a higher CS and CF.

4.2.2. Defects type impact on the CS.

The association of Cs to each defect type is performed as follows: 1) We count the number of classes participating in a specific type of aesthetic defect. 2) For each defect type we asses whether the participating classes have greater CS variation comparing with classes participating in another type of defect.

Our computation is a manual process done via the GitHub platform. We go through all the versions of each application considering the number of commits, and the change-size for each class. All the experimented data are available URL (b).

We do not consider the impact of defects type on the change frequency in this RQ, because we believe that the correlation between the variables (type, CF) is not logical since the defect will impact the source code that is related to the applied change-size. However, we can relate the CF to defects type when it comes to developers quality. Basically, the CF will give us an indication on the change quality if the defect persists from a given version to the next one. We will target this hypothesis in our future work.

The Context consists of the change history of 5 selected Android applications from the Google Play Store. **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 has been 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 4.1 rating 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 4.1 rating on Google Play store. Table 2 presents the applications technical characteristics.

Table 2: Characteristics of the tested android applications.

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

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.

Figure 3, reports the box plot of the number of aesthetics defects instances in our analyzed android applications. For each aesthetic defect, we aggregate its occurrence number in all the releases of the five applications. The box plot bring out significant differences in the diffuseness of aesthetics defects. There are defects like Incorrect appearance of widgets, InCohesion of MUI, Incorrect data presentation , and Imbalanced MUI are weakly frequent in the applications. For instance, we found the highest number of Imbalanced MUI instances is 11 in the V3.2.0a-beta of

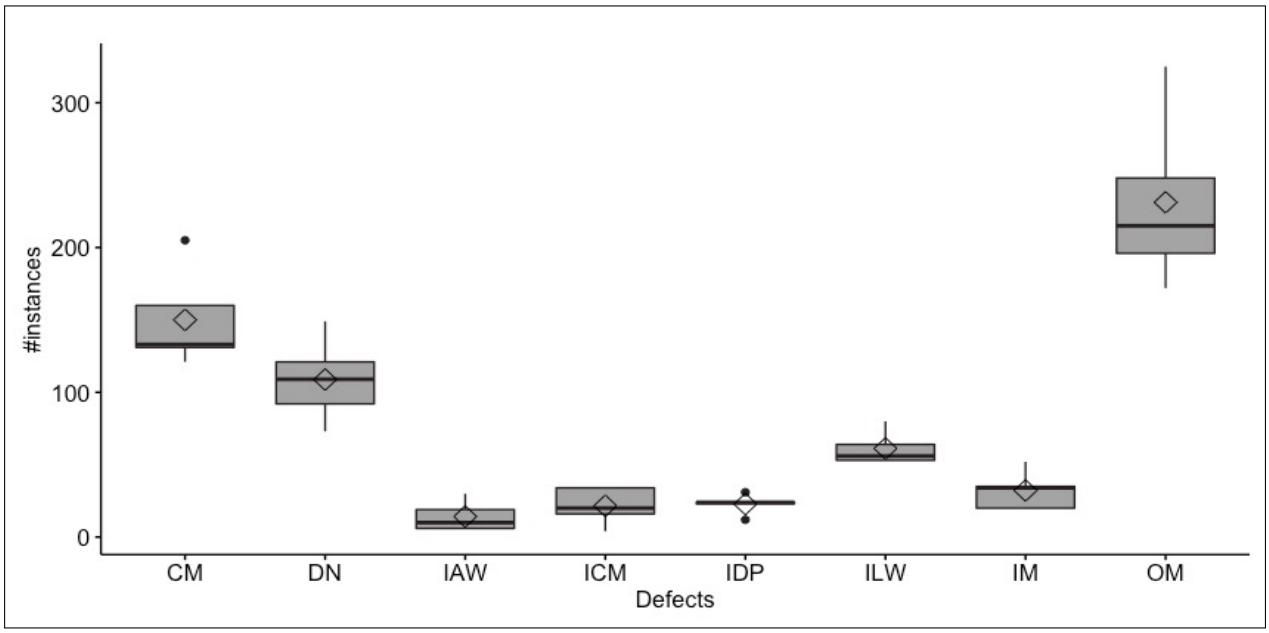

Figure 3: Number of Aesthetics defects instances in the analyzed applications.

Lightning app, leading to 0.2 probability of a class getting infected by this defect. However, it does exist in 69.6% of the releases. Noting that our study resulted in categorizing the Imbalanced MUI as the third defect having the highest correlation with infected classes change-size. Therefore, the box plot results highlight that although the IM defect is harmful its distribution across the applications is limited. Incorrect appearance of widgets is also poorly defused. 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 were 65 out of 101 (65%).

Other aesthetics defects are in opposite quite diffused. For example, we found instances of the Overloaded MUI in 100% of the analyzed releases, with a probability of 0.71% of appearing in a class. In particular, weather V3.0.1 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.

Another proliferated aesthetic defect is the Complicated MUI, that affects 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. Finally, the Difficult navigation with 96.42% of affected releases with a probability of 0.41 of being affected.

Interestingly, we noticed that Weather application has the highest number of three defects: Incorrect layout of widgets, Overloaded MUI, and Complicated MUI. We downloaded two releases with big time-frame (time of release) V2.0 (2018-03-19), and V1.1 (2016-11-13) from F-droid platform to understand the reasons behind the presence of these defects. We chose to show the evolution of one User interface (Weather overview UI) for sake of clarity as seen in Figure **??**. We evaluated these two UIs by PLAIN and we got 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 different big quantity of elements making it look charged. Al thought, the UI V2.0 has a minimum and structured elements, the non-inter-relatedness of layouts makes it feel overloaded. Although, 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 persist. We can explain this persisting and the reason behind having the OM and CM as the most frequent defects by the nature of the implicated structural metric, where this latter takes as parameters in the nominator the result of all other metrics.

Thus, considering more characteristics of the UI, making it so delicate.

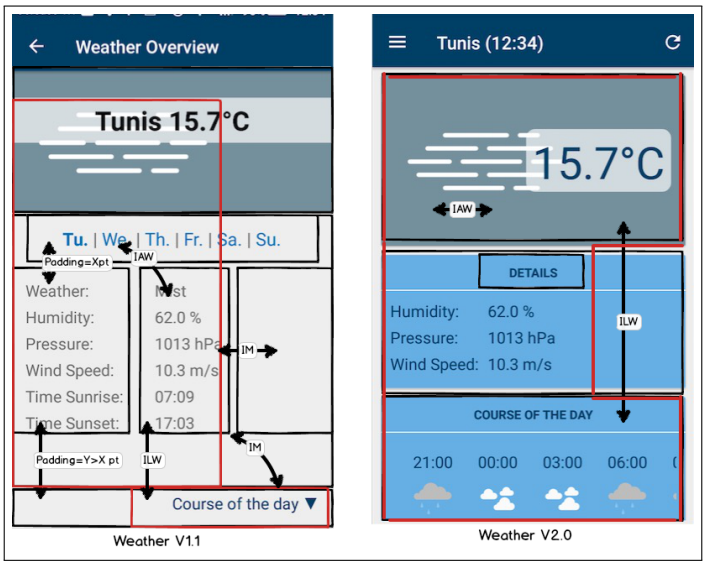

Figure 4: Weather Overview UI in two consecutive releases V1.1 (left), and V2.0 (right)

4.3. 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.

RQ1: *Is the application more susceptible to GUI defects through its several updates?* This is a preliminary investigation that aims to control the existence of aesthetics defects along the releases by testing the following hypothesis.

$H1_0$: The application is more susceptible to GUI defects through its evolution.

RQ2: *Are infected GUI classes more change-prone than non-infected GUI classes?* We are interested to see whether developers mostly focus on refactoring infected classes over time by testing the following null hypotheses.

$H2_0$: The CF of infected GUI classes is equal to the CF of non-infected GUI classes.

RQ3: *Do infected GUI classes require more LOC change than non-infected classes?* 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.

$H3_0$: The CS of infected GUI classes is equal to the CS of non-infected GUI classes.

RQ4: *Are particular types of GUI defects responsible for the change-size?* We examine the influence of particular types of GUI defects on the CS by testing this null hypothesis:

$H4_0$: GUI classes infected with a given type of GUI defect have not more change-size in comparison to other classes.

4.4. 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: RQ2 and RQ3, we use the Mann-Whitney U-test to understand whether the change frequency (RQ2), and the change-size (RQ3) differ based on class type. i.e., our dependent variables would be the ”CF” for RQ2 and ”CS” for RQ3.

RQ4, The change-size is the number of LOC changes that a class underwent (i.e., addition/removal/modification) between version v(n) and its anterior v(n-1) excluding comments and blank lines. For each release of the application, each class is characterized by its change-size and the total number of defects in which it participates.

Independent variables: RQ2 and RQ3, would be the ”class type”, which has 2 groups (infected GUI and noninfected GUI).

RQ4, we extract the existence of 8 types of GUI defects. Each variable $G_{c,d,v}$ refers to how many instances of a defect d a GUI class c has in a version v.

4.5. Measurement Method

In RQ1: In order to track the number of GUI smells over the different releases, we consider the evaluation of PLAIN tool for each app release.

In RQ2 and RQ3: we have relied on the calculation of the change frequency and the change-size given. (1) 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≤ CS. (2) the change-size (CS): it is how many lines of code have been changed within a class in a release. i.e., (addition/removal/modification) CS≥CF

To test our hypotheses, the change frequency (CF), and the change-size (CS) were calculated for the infected and noninfected GUI classes.

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

$$CS(C_t) = \frac{\sum_{c=1}^{n} CSIZE(C_t) * 100}{LOCC(C_t)} \tag{2}$$

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;

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.

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.

In RQ4: our goal is to deduce the impact of specific kinds of GUI defects on the CS of smelly classes.

We modeled the correlation of existent GUI defects with CS using logistic regression model Khomh et al. (2012). By definition, this model relies on the definition of dependent and independent variables. Dependent variables are naturally containing a binary decision 0, 1, for example, they represent whether there is a change or not. The multivariate logistic regression model is based on the formula Hosmer and Lemesbow (1980):

$$\pi(X_1, X_2, ..., X_n) = \frac{e^{C_0 + C_1.X_1 + ... + C_n.X_n}}{1 + e^{C_0 + C_1.X_1 + ... + C_n.X_n}} \tag{3}$$

where:

Xj represent modeled phenomenon characteristics, particularly the number of defects of kind j an interface has. i.e., when the model is applied to the GUI class c of release v.

j are the model coefficients.

π is a value on the logistic regression curve.

As the value of π close to 1 signifies a higher chance of a GUI undergoing a potential change and so H04 is rejected.

Afterward, we consider each type of defect across the analyzed versions of each app, and we report how many times the p-values extracted from our model were found to be significant.

5. Statistical results of research questions

Results of RQ1: We calculate the percentage of aesthetic defects along the application evolution. In the Figures 5-9, 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 GUI classes per release.

$H1_0$: *The application is more susceptible to GUI defects through its evolution.* The analysis of GUI defects occurrence frequency through the five apps evolution, denotes that the applications did not have more smells over updates. Referring to graphs 5 to 9, we can conclude that this statement is totally dependent on the number of infected GUI classes in each release. It is noticeable that for the five applications, the behavior of aesthetics defects curve did follow the rate of the infected classes curve. So, we fail to accept (H1).

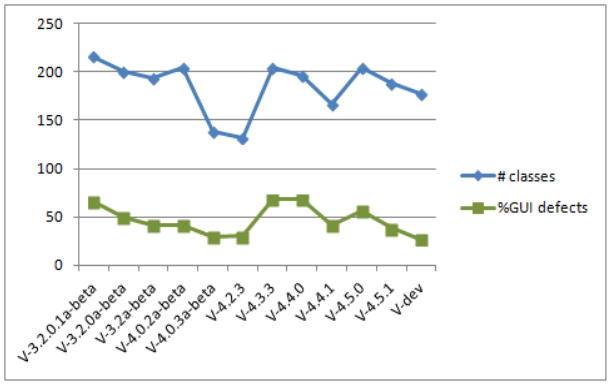

Figure 5: Defects density of Lightning.

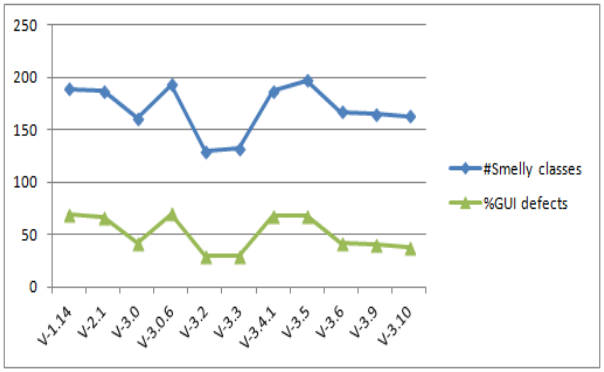

Figure 6: Defects density of Mattermost.

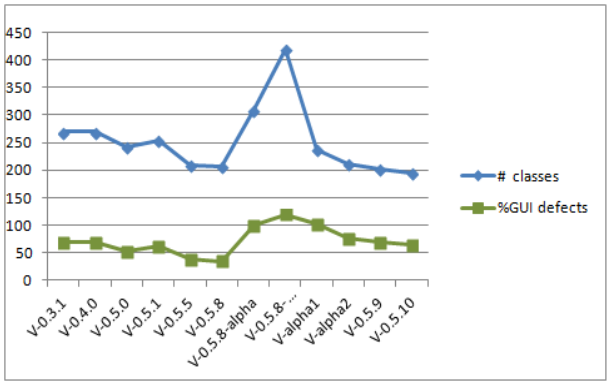

Figure 7: Defects density of Openlauncher.

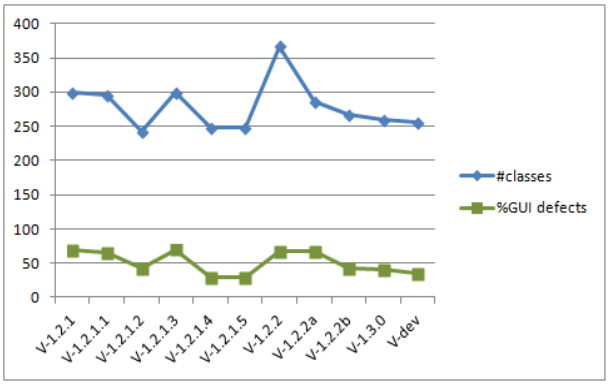

Figure 8: Defects density of Reddit.

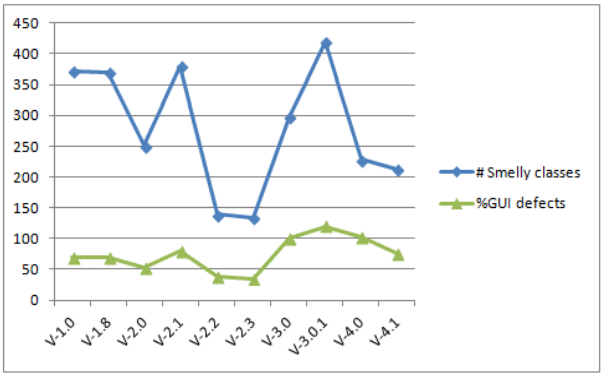

Figure 9: Defects density of Weather.

Results of RQ2, RQ3: Tables 3-7 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 GUI defects relate to an increase in the change frequency and change-size. In the result tables presented below, n represents the samples size, M for the median, sd for the standard deviation, U for the calculated U-test, and P for P-value.

$H2_0$: The CF of infected GUI classes is equal to the CF of non-infected GUI classes.
The hypothesis is rejected by all five apps. Tables 3-7 show that the medians of CF for infected GUI Classes are remarkably (3-25-30-30-27 times) higher than for non-infected GUI classes. These results show that each infected GUI class line of code (LOC) is changed more often than non-infected classes.

$H3_0$: The CS of infected GUI classes is equal to the CS of non-infected GUI classes.

Tables 3-7 show that the size of changes performed per line of code in the infected GUI classes is, noticeably, higher than for non-infected classes. For example, in Mattermost app, the change-size for infected GUI classes is on average 60 times higher than non-infected GUI classes. The alternative hypothesis is supported by all five applications.

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	

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	

Results of RQ4: Table 8 summarizes the results of a logistic regression at a significance level of 0.5 for the relation between kinds of GUI defects and change-size of a class. The % cell in the table represents the number of a GUI defect with the percentage of classes significantly correlating with this latter. The gray shaded cell shows a p-value inferior to 0.5, which means the given defect substantially correlates with the change-size, and that we reject

the null hypotheses. We highlight the intersection of defects correlating with more than 50% of classes with a pink shade.

$H4_0$: GUI classes with particular kinds of GUI defects have not more change-size than other classes.

For example, the cell at the intersection of the "Lightning" app with the "Incorrect layout of widgets" defect shows a significant impact of the defect on the CS with a 0.0033. This indicates that 79% of classes participating in "Incorrect layout of widgets" are more change-prone than other non participating infected classes. Moreover, we do notice that 75% of the pink shaded cells have a strong correlation < 0.1 whenever we have more than 50% of correlating classes. Which we claim that the involved defects are more severe (requiring more modifications) than the other defects.

From Table 8, we can reject $H4_0$ for some defects that are significantly correlated to change-size in at least 60% of classes. Following our analysis, only "Overloaded MUI" has a significant impact on change-size in all applications with more than 60% of participating classes. Thus, classes participating in this defect are more likely to change than classes participating in others defects. We claim that being the "Overloaded MUI" the

Table 8: The percentage of infected classes where each defect significantly correlates with change-size along with the number of defect in the correlating class

Defects	**change-size**									
	Lightning		Mattermost		Reddit		Open launcher		Weather	
	%	P-val	%	P-val	%	P-val	%	P-val	%	P-val
Imbalance of MUI	45(60%)	0,3079	77(83%)	0,1344	65(59%)	0,0239	107(79%)	<0,0001	54(34%)	0,3409
Incorrect layout of widgets	50(79%)	0,0035	24(35%)	0,3537	34(30%)	<0,0001	113(83%)	0,2783	38(24%)	0,9264
Difficult navigation	14(26%)	0,0543	3(12%)	0,8725	50(45%)	0,0465	50(37%)	0,0006	100(64%)	<0,0001
Incorrect data presentation	84(62%)	0,4309	5(16%)	0,2231	79(71%)	0,003	122(90%)	0,7251	31(20%)	0,6720
Ineffective appearance of widgets	45(59%)	0,2450	6(17%)	0,9978	57(51%)	0,0863	99(73%)	0,1125	75(48%)	0,2308
InCohesion of MUI	120(70%)	0,1127	7(18%)	0,8667	43(39%)	0,0060	50(37%)	0,4876	49(31%)	0,0015
Overloaded MUI	65(83%)	0,0069	62(70%)	0,0019	82(74%)	<0,0001	93(68%)	<0,0001	104(67%)	0,0235
Complicated MUI	59(73%)	0.0003	78(87%)	0,4374	45(40%)	0,1846	48(35%)	0,0007	64(41%)	0,0007

most defect correlating with the change-size, it is seen severe from an end-user point of view. We believe that an overloaded UI will hinder the user from easily and successfully interacting with the application. Other defects have a significant impact on the change-size of only a subset of applications and infected classes like Difficult navigation, InCohesion of MUI, and Complicated MUI in Reddit, OpenLauncher, and Weather. Incorrect data presentation and Incorrect layout of widgets in Mattermost.

6. Discussion

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

6.1. Experiment results analysis

RQ1: Referring to the graphs 5-9, we can deduce that the curves of infected GUI classes and aesthetics defects evolve at the same pace. At this stage of the study, the intensity of the correlation can be only justified based on the behavior of bad defects in relation with the number of infected classes (GUI), and it is 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 vise 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.

RQ2 and RQ3: From Tables 3-7, it can be noticed that infected and non-infected GUI classes were significantly different regarding their change frequency and change-size. We showed that CS and CF of infected classes is 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.

RQ4: From Table 8, we conclude that there is a relation between kinds of GUI defects with the classes change-size but not for all defects and not for all the applications.
We fail to accept $H4_0$ for the five applications for the Overloaded MUI defect. Classes participating in such defect are more change-prone than any other classes, possibly because these classes are the most related to the launching activities, hence, most complex and operative. Thus, they are more likely to be changed to fix the GUI defects, and consequently, faults are more expected to be present. Yet, we fail to reject the hypotheses for the "Incorrect data presentation" for the OpenLauncher application, where 90% of classes participate in the defect, yet no correlation?. We recall that the Openlauncher app is an Android launcher application that lets users customize their home screen (adding widgets, launch apps, add animations, transitions, etc.). If we review in detail the nature of a launcher application, we will see that it does not support the kind of an incorrect data presentation GUI defect. Because there is no auto-display of information on the screen by the application. It is all in the hands of the user on how to display his own chosen content on the UI. Consequently, we indicate that in this application the number of defects has no interference in the change-size of classes. However, we cannot yet generalize the conclusion at this stage without a deep empirical test.

Table 9 generalizes the findings of the logistic regression for the five applications. We rated the defects based on the sum of the number of classes that significantly correlate with defects in the five applications.

In a scale of five applications, we found out that "Overloaded MUI", "Complicated MUI," and "Imbalance of MUI" are the top three defects having an impact on the change-size of the participating classes consecutively. In a scale of one application, the ranking might vary. For example, we ranked the "Difficult navigation" defect as number 7 in the general scale is one of the least defects that correlate with classes. However, in the Weather application, 64% of classes participate in this defect with the strongest correlation of <0.0001. This result shows that the defect is impacting the CS of classes hugely. i.e., developers mostly tend to fix the "Difficult navigation" defect. Thus, the defect is considered the most severe in that context.

We claim that GUI defects might vary of specificities from an MUI to another.i.e., if an Overloaded MUI defect is detected, it does not necessarily mean that developers will rush to fix this defect. In some cases not correcting the most

severe deficiency on the CS and fix another defect in favor of the latter is the preferable way. As we have shown in the illustrative example section, adjusting the difficult navigation issue as being not severe on the change-size compared with Overloaded MUI defect will correct the problem. However, for an aesthetic reason, it is favorable to fix the Overloaded MUI issue and do extra workload to produce an appealing UI.

Table 9: Summary of our findings for RQ4.

Rank	**Defects**	**Lightning**	**Mattermost**	**Reddit**	**Open Launcher**	**Weather**	**%Class**
1	OM	X	X	X	X	X	75%
2	CM	X	X	X	X	X	61%
3	IM	X	X	X	X	X	56.5%
4	IAW	X	—	X	X	X	60.5%
5	ILW	X	X	X	X	—	56.5%
6	ICM	X	—	X	X	X	50%
7	DN	X	—	X	X	X	45%
8	IDP	X	X	X	—	—	43.5%

6.2. Implications for research

Based on recent studies on GUI 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 GUI defects are detected. Taking into account the severity of defects types provides guidance to software engineers on improving their maintenance operations. Furthermore, It would be interesting to investigate the severity of these defects types on two levels: 1) the severity of the defects on a class change-proneness.i.e., how much LOC changes a defect type requires to fix it. 2) the severity on the user satisfaction. From a user point of view, which defect is more likely to deteriorate the usability level of the interface. This hypotheses can be further addressed 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 GUI 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-proneness and change-size as two main measurements we are correlating with the existence of defects. In fact, many factors may be also 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 that may trigger a wider set of changes is the density of the interface, i.e., UIs with more services tend to be 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 contributors 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

It questions the generalizability of our findings. We have purposely chosen 5 different Android projects 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 reported 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 GUI classes, which resulted in a robust significant difference for the five applications. Following, we investigated the impact of defects types on the change-size. The outcomes show that some particular defects contributed to more change-size than others.

The debate here concerns "When a defect should be removed?. Some designers might choose to fix the defect with the minimum impact on change-size. Others might prioritize the visual aesthetic attractiveness of the user interface, and choose to fix the defect with the higher impact on the change-size. As designers seek to provide seamless UIs, the higher probability will be for the second choice. However, the point that arises here is: If a UI has, for instance, the "Overloaded MUI" defect revealing a high components density, with a balanced, not complicated, and comprehensible UI, should we consider this defect as a harmful defect?. One of our future directions will be to work on the restructuring of the "Overloaded MUI" aesthetic defect. We tend to see whether reducing the number of components in a UI will increase or not its visual attractiveness and usability?.

In a high-level study, we showed that the number of aesthetic defects and infected GUI classes do match all along the app's evolution. In future work, we want to decipher this matching in a deep-level. To generalize our findings, we plan to enlarge the scale of experimented applications. 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.

Our defects severity evaluation of the change-size is project dependent, which means it might vary depending on the parameters and the architecture of each application. Except that we have seen the same behavior of defects impact mostly across all projects. Still, we plan on extending our studied projects to generalize our findings.

References

, a. https://android.jlelse.eu/apple-vs-android-a-comparative-study-2017-c5799a0a1683.

, b. https://github.com/mabroukachouchane/correlation.

, c. https://www.abiresearch.com/press/ android-will-account-for-58-of-smartphone-app-down,.

, d. https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/.

Bastien, J.C., Scapin, D.L., 1995. Evaluating a user interface with ergonomic criteria. International Journal of Human-Computer Interaction 7, 105–121.

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: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, IEEE. pp. 56–65.

Bavota, G., Qusef, A., Oliveto, R., Lucia, A., Binkley, D., 2015. Are test smells really harmful? an empirical study. Empirical Softw. Engg. 20, 1052–1094. doi: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. arXiv preprint arXiv:1703.10674 .

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

Hosmer, D.W., Lemesbow, S., 1980. Goodness of fit tests for the multiple logistic regression model. Communications in statistics-Theory and Methods 9, 1043–1069.

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

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 URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.150.1292&rep=rep1&type=pdf, doi:10.1.1.150.1292.

Khomh, F., Di Penta, M., Guéhéneuc, Y.G., Antoniol, G., 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empirical Software Engineering 17, 243–275.

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, 1120–1128. doi:10.1016/j.jss.2006.10.018.

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

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

Myers, A.C., 1995. Bidirectional object layout for separate compilation, in: ACM SIGPLAN Notices, ACM. pp. 124–139.

Ngo, D., Teo, L., Byrne, J., 2000. Formalising guidelines for the design of screen layouts. Displays 21, 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, 50–69.

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

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

Park, J., Han, S.H., Kim, H.K., Cho, Y., Park, W., 2013. Developing elements of user experience for mobile phones and services: survey, interview, and observation approaches. Human Factors and Ergonomics in Manufacturing & Service Industries 23, 279–293.

Sheskin, D.J., 2003. Handbook of parametric and nonparametric statistical procedures. crc Press.

Silvennoinen, J., Vogel, M., Kujala, S., 2014. Experiencing visual usability and aesthetics in two mobile application contexts. Journal of usability studies 10, 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.

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, ACM, New York, NY, USA. pp. 4–15. doi:10.1145/2970276.2970340.

Türkyilmaz, A., Kantar, S., Bulak, M.E., Uysal, O., et al., 2015. User experience design: Aesthetics or functionality?, in: Managing Intellectual Capital and Innovation for Sustainable and Inclusive Society: Managing Intellectual Capital and Innovation; Proceedings of the MakeLearn and TIIM Joint International Conference 2015, ToKnowPress. pp. 559–565.

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

Zen, M., Vanderdonckt, J., 2014. Towards an evaluation of graphical user interfaces aesthetics based on metrics, in: Research Challenges in Information Science (RCIS), 2014 IEEE Eighth International Conference on, IEEE. pp. 1–12.