Domain Matters: Bringing Further Evidence of the Relationships among Anti-patterns, Application Domains, and Quality-Related Metrics in Java Mobile Apps

Mario Linares-Vásquez[1], Sam Klock[1], Collin McMillan[2], Aminata Sabané[3], Denys Poshyvanyk[1], Yann-Gaël Guéhéneuc[3]

[1]Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185
{mlinarev, skkloc, denys}@cs.wm.edu

[2]Department of Computer Science
University of Notre Dame
Notre Dame, IN 46556
cmc@nd.edu

[3] Département de génie informatique
École Polytechnique de Montréal
Montréal, Québec, Canada
{aminata.sabane, yann-gael.gueheneuc}@polymtl.ca

ABSTRACT

Some previous work began studying the relationship between application domains and quality, in particular through the prevalence of code and design smells (*e.g.*, anti-patterns). Indeed, it is generally believed that the presence of these smells degrades quality but also that their prevalence varies across domains. Though anecdotal experiences and empirical evidence gathered from developers and researchers support this belief, there is still a need to further deepen our understanding of the relationship between application domains and quality. Consequently, we present a large-scale study that investigated the systematic relationships between the presence of smells and quality-related metrics computed over the bytecode of 1,343 Java Mobile Edition applications in 13 different application domains. Although, we did not find evidence of a correlation between smells and quality-related metrics, we found (1) that larger differences exist between metric values of classes exhibiting smells and classes without smells and (2) that some smells are commonly present in all the domains while others are most prevalent in certain domains.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

General Terms

Measurement, Design, Theory

Keywords

Anti-patterns, Internal metrics, Software quality, Domain categories, Java Mobile Edition

1. INTRODUCTION

An open (and controversial) question in software engineering relates to the relationship between software practices and software quality. The concept of design patterns—structured, reusable solutions to recurring design problems, typical example of good practices, was introduced in software engineering by the "Gang of Four" (GoF) [23]. Following the GoF, since their introduction, the conventional lore among both researchers and practitioners has been that the use of design patterns improves some quality characteristics, *e.g.*, maintainability, by making source code easier to understand, more stable, and with fewer faults. Conversely, code smells and anti-patterns—recurring solutions to code and design problems, *i.e.*, poor practices, are assumed to decrease quality [11]. In general, code smells are symptoms of the presence of anti-patterns in the code. In the following, we use the word "smells" to refer to both code smells and anti-patterns, as in previous work [40].

However, the relationships among smells, application domains, and quality have been little studied so far. In general, following Zhang and Budgen and their mapping study [56] for design patterns, we argue that much of the research on the presence of smells and their relationships to application domains and quality-related metrics has been small-scale and frequently anecdotal: experience reports, case studies, developers' opinions. A notable exception is the work by Fontana *et al.* [22], which studied the relationships between anti-patterns and application domains and concluded that no relationship could be found in the curated *Qualitas Corpus*, contrary to the lore. While this previous research work is useful for practitioners and researchers by discussing the relationships between anti-patterns and application domains, we believe that the following general question remains unanswered: **"Are applications in different domains negatively impacted by smells in different ways?"** The answer to this question, coupled with concrete knowledge on the relationships that smells have with software quality, is of interest to researchers and practitioners. A related question—the relationship between domain and quality—also has interpretive significance. On the one hand, if the prevalence of smells and software quality significantly varies by domains, then it is important to report such variations and to understand why they occur: practitioners should be wary of the smells most prevalent in their application domains while researchers should take into account the domains of the applications under analysis in their algorithms. Also, different prevalence among domains would show that opportunistic programming [7] and applications developed by domain experts (who may not be software experts) can have poor quality and, thus, should be used with caution. On the other hand, the absence of any relationship between domain and quality would imply that the presence of smells does degrade quality *per se* and would warrant further studies on their impact.

With the purpose of providing answers to the general question, we follow previous works and study the relationships among smells, application domains, and quality on a large scale. We obtain empirical results on the relationships between the presence of smells, source-code metrics, and application domains, using two correlation measures (Spearman and Kendall) and the Cliffs's *d* effect size [24]. We use DECOR [40] to identify occurrences of smells and POM [26] to compute quality-related metrics from the bytecode of 1,343 Java Mobile Edition Applications belonging to 13 domains in a closed-source repository. We assess whether the presence of smells had any relationship with the values of the metrics and the domains. We show that (1) some smells are common in all the domains while others are more common in certain domains and (2) the presence of smells increases the values of fault-related metrics, although there is no evidence of a correlation between smells and metrics. Thus, we contribute to the researchers' and practitioners' knowledge about the relationships among smells, application domains, and quality.

2. RELATED WORK

Object-oriented metrics (OO metrics) have been widely used to assess the internal quality of applications. Additionally, quality models define quality attributes as aggregation of lower-level attributes and code-level metrics [3; 4; 30]. Several works used OO metrics as indicators of quality, such as fault-proneness, testability, and change-proneness [1; 5; 6; 8; 10; 12; 15; 16; 21; 29; 32; 36; 41; 47; 49-52; 55; 57]. They mostly focused on the well-known Chidamber and Kemerer (CK) metric suite of OO metrics [17]. Several other works used metrics to describe and detect occurrences of design patterns and–or anti-patterns, such as the works on detection strategies by Antoniol *et al.* [2], Lanza and Marinescu [35], and Marinescu [39].

Since their introduction in the field of software engineering, smells have been the subject of much work, either for their definition/formalization, the identification of their occurrences, or their categorization. However, previous work has not yet verified the relationships among smells, application domains, and quality on a large scale. Li and Shatnawi [37] analyzed the relationship between six anti-patterns and class fault-proneness in three versions of Eclipse. Bug reports severity was used as a proxy for fault-proneness. *Shotgun Surgery, God Class*, and *God Methods* were positively related to fault-proneness. Although we did not use fault-related data to identify the impact of smells in software quality, we used OO metrics as a proxy for quality and looked for occurrences of 18 anti-patterns in 1,343 apps.

Khomh *et al.* [33] performed a study on the impact of 13 anti-patterns in 54 releases of ArgoUML (0.10.1 – 0.26.2), Eclipse (1.0 – 3.3.1), Mylyn (1.0.1 – 3.1.1), and Rhino (1.4R3 – 1.6R6), to identify the relationship between smells and class change- and fault-proneness as well as between the occurrences of the smells and the sizes of the classes having these smells. In our study, we complement the findings of Khomh *et al.* [33] by analyzing the impact of smells on fault-related metrics.

Romano *et al.* [45], in a study of 16 Java systems, concluded that classes affected by smells are more change prone, in particular classes participating in the *ComplexClass*, *SpaghettiCode*, and *SwissArmyKnife*. According to our results, classes participating in these three smells have larger values of LCOM5 compared to classes without them. These results corroborate Romano *et al.* [45] findings: classes with low cohesion implement several and unrelated functionalities and then are change-prone.

Yamashita and Moonen [54] and Yamashita and Counsell [53] analyzed the impact of 12 smells on the maintainability of four Java-based systems. Smells were detected using commercial tools that implemented the detection strategies by Lanza and Marinescu [35] and Marinescu [39]. The main conclusion of both studies is that smells are present in classes with maintainability issues; therefore smells could be used for assessing the maintainability of a system.

Jaafar *et al.* [31] showed that, in several releases of ArgoUML, JFreeChart, and XercesJ, classes with smells are more fault-prone than other classes and that this observation is also true for classes with binary-class relationships with smelly classes. In our study, we did not analyze static relationships between smelly classes and other classes.

Olbrich *et al.* [42] performed an empirical study that showed that *God* and *Brain classes* were more change and fault prone than other classes. However, when normalizing class metrics with respect to size, they observed that *God* and *Brain classes* were less change and fault prone than others. They concluded that *God* and *Brain classes* are not necessarily harmful because they contain as much functionality per line of code as others.

Guo *et al.* [28] performed an empirical study to assess the benefit of including tailored domain-specific rules in metrics-based smell detection techniques. These specific-domain rules are obtained from domain experts and used to refine smell definitions by modifying some thresholds and–or rules. These tailored rules make the detection results more accurate.

Fontana *et al.* [22] performed an empirical study to identify the most frequent smells in systems of different domains and investigate whether there are correlations between smells and metrics. The results showed that the most frequent smells are *Duplicate Code*, *Data Class*, *God Class*, *Schizophrenic Class*, and *Long Method,* but no specific domain contains more smells of a particular kind. The authors used the applications in the curated *Qualitas Corpus* for their study. We follow this study but argue that the size of the corpus is too small to observe an effect. Therefore, we design our study to use more than a thousand applications belonging to 13 domains.

Palomba *et al.* [43] proposed an approach for detecting five different code smells (*Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy)*, by exploiting change history information mined from versioning systems. In particular they analyzed the versioning systems of eight software projects written in Java.

3. EMPIRICAL STUDY DESIGN

The *goal* of this study is to identify if a relationship exists between the presence of smells and quality-related metrics and between application domains and the presence of smells in Java Mobile Applications. The *context* consists of 1,343 Java Mobile Applications from ShareJar[1] that belong to 13 domains (categories in the context of ShareJar) and the *quality focus* is the quality of the applications measured by 48 OO metrics.

[1] The Java Mobile Edition apps were downloaded in 2011 from http://www.sharejar.com. However, by the time we wrote this paper the domain was on sale and the download access was deactivated. Therefore, we could share the JAR files upon request.

TABLE 1. ShareJar categories represented in the data

Category	Label	#Apps	Category	Label	#Apps
Chat&SMS	A	290	Music	H	77
Dictionaries	B	58	Science	I	30
Education	C	163	Utilities	L	364
Free Time	D	211	Emulators	M	66
Internet	E	333	Programming	N	20
Localization	F	32	Sports&Health	O	68
Messengers	G	96	Uncategorized	U	650

Table 1 lists the number of Java Mobile Applications per category[2]. We used free Java Mobile Applications because the bytecode (JAR files) of the apps was publicly available and ShareJar allowed us to download a set of apps that belongs to several application domains.

The software repository that we used as representative of Java Mobile Applications, ShareJar, like other software repositories and code-search engines, provides a predefined list of categories that represent application domains. Therefore, we used ShareJar *software categories* as representative of application domains. Developers associate applications to categories to make the searching and browsing easier. Although the category of an application depends on the features provided by the application, developers must often choose from a predefined list. In some cases, the list does not reflect the developers' intention; in some other cases, developers may have chosen by mistake categories that do not represent their applications. We assume that developers "correctly" assign categories to their applications and that the categories available in ShareJar comprehensively reflect the domain of the applications stored therein. Thus, we use categories according to the mappings supplied by ShareJar.

3.1 Research Questions

Following our general question in Section 1, we ask the following research questions:

- **RQ$_1$**: What relationship, if any, exists between the presence of anti-patterns and quality-related metrics in Java Mobile Applications? (Are certain anti-patterns more likely to impact software quality, in particular fault-proneness?)
- **RQ$_2$**: What relationship, if any, exists between the presence of code smells and quality-related-metrics in Java Mobile Applications? (Are certain code smells more likely to impact software quality, in particular fault-proneness?)
- **RQ$_3$**: What relationship, if any, exists between software categories in ShareJar and the presence of anti-patterns in Java Mobile Applications? (Are certain smells more likely to appear in particular categories?)
- **RQ$_4$**: What relationship, if any, exists between software categories and quality-related metrics in Java Mobile Applications? (Are applications in certain categories more likely to be characterized by certain metric values?)

The **dependent variable** for **RQ$_1$**, **RQ$_2$**, and **RQ$_4$** is represented by the values of the OO metrics computed on the Java Mobile Applications. The **dependent variable** for **RQ$_3$** is represented by the occurrences of smells in the Java Mobile applications.

The **independent variable** for **RQ$_1$** is the number of occurrences of *anti-patterns* in the applications; for **RQ$_2$**, it is the number of occurrences of *smells*, *i.e.*, roles in *anti-patterns*; for **RQ$_3$** and **RQ$_4$**, it is the categories of the applications.

With **RQ$_1$**, we want to analyze whether some metrics are indeed correlated with the presence of occurrences of anti-patterns (one or more). **RQ$_2$** provides a complementary, interesting answer to that of **RQ$_1$**, focusing on the code smells (anti-pattern roles) rather than on the anti-patterns. Similarly with **RQ$_3$**, we want to identify if anti-patterns may be more or less prevalent depending on the category. Finally, with **RQ$_4$**, we expect metric values to change across categories because each category may impose designing and programming styles, which will be captured by some of the metrics.

3.2 Data Extraction Process

Our data is from the ShareJar repository, a collection of applications for mobile devices. The applications in the repository belong to 13 categories (some of the applications are uncategorized), which are reported in Table 1. All of the applications are written in Java Mobile Edition and take the form of bytecode stored in JARs. Of all the applications that we collected from ShareJar, we were able to analyze 1,343. Unfortunately, ShareJar does not keep track of the different versions of a given JAR. Therefore, we cannot report an analysis of the evolution of the JARs in time in terms of occurrences of smells and internal metric values, as some previous work did on design patterns and smells [20; 33; 34] but with smaller numbers of systems. Also, ShareJar does not provide a bug-reporting tool, so we cannot analyze the fault-proneness of the studied applications. Therefore, we rely on previous work that studied the relationships between metrics and fault-proneness to discuss in Section 5 the presence of smells and its relationship to class fault-proneness.

An important property of the analyzed applications is that some of them belong to more than one category. The overlap introduces the possibility of correlations in the results among categories. We indirectly study and reject any effect of the overlapping categories on the results of our study when answering **RQ$_3$**. Our study yields us to observe that there are significant differences among the categories in the frequency distribution of anti-patterns. These differences are significant and let us conclude that any observation made on a category cannot be due to the overlap because such observation would then be similar to that made for some other overlapping category, which is not the case as shown by our answer to **RQ$_3$**. Therefore, even in the presence of overlap, the obtained results are sufficient to examine *whether or not* relationships with respect to category exist. Also, 650 of the applications are not categorized; we exclude their JARs from our analysis for the research questions related to software categories (**RQ$_3$** and **RQ$_4$**).

We rely on tools belonging to the Ptidej[3] framework to create models of the applications, identify binary class relationships between model entities representing classes and interfaces [27], compute OO metrics from these models using the POM framework [26], and identify occurrences of smells in these models using the DECOR framework. POM allows computing 48 OO metrics, which values are reported in the online appendix[4] and definition can also be found on-line[5]. DECOR [40] has been reported to have reasonable precision and has been used in several

[2] Each application may belong to one or more category.

[3] http://www.ptidej.net

[4] http://www.cs.wm.edu/semeru/data/ICPC14_antipatterns/

[5] http://wiki.ptidej.net/doku.php?id=pom

previous studies [20; 33; 34]. It uses rule cards to specify each code smell or anti-pattern based on properties of the classes according to their lexicon (*i.e.*, names), structure (*e.g.*, classes using global variables), and internal attributes using metrics. As shown by Khomh et *al.* [33], the use of metrics in smell detection does not lead to a correlation between code smells and anti-patterns and metrics, because their detection involve other class properties, such as binary class relationships. The 18 smells available in DECOR are defined on-line[6] and are: *AntiSingleton (AS), BaseClassKnowsDerivedClass (BCKDC), BaseClass-ShouldBeAbstract (BCSBA), Blob, ClassDataShouldBePrivate (CDSBP), ComplexClass (CC), FunctionalDecomposition (FD), LargeClass (LC), LazyClass (LZC), LongMethod (LM), LongParameterList (LPL), ManyFieldAttributesButNotComplex (MFABNC), MessageChain (MC), RefusedParentBequest (RPB), SpaghettiCode (SC), SpeculativeGenerality (SG), SwissArmyKnife (SAK),* and *TraditionBreaker (TB).*

3.3 Analytical Method

We formalize the concept of a relationship between the presence of anti-patterns and the values of the OO metrics as the association where the occurrences of anti-patterns affect the distribution of particular OO metrics. The presences of anti-patterns as well as the metric values are all computed at the class level. DECOR and POM use classes at their unit of interest to compute OO metrics; the former reports whether a particular class in an application plays some role(s) in some anti-pattern(s) while the latter reports the set of metric values associated with each class. We then use the role(s) played by a class in some anti-pattern to state whether the class belongs to that anti-pattern—indifferently of the number of (possibly different) roles that the class may play in this anti-pattern. Similar to previous works, we observed that classes may play no role, one role, or more than one role in anti-patterns. Consequently, we cannot use a dichotomous variable to describe the role played by a class in an anti-pattern or whether the class exhibits or not the anti-pattern (no matter its role).

For $\mathbf{RQ_1}$ and $\mathbf{RQ_2}$, we decided to use correlation coefficients because these are descriptive statistical measures that demonstrate the strength or degree of a monotone association between two variables [48]. We used correlation coefficients to measure different aspects of the dependence structure in the data; in particular, without normality assumptions, we measured the strength and direction of non-linear relationship using Spearman and Kendall coefficients. To categorize the strength of a direction, we used the guidelines provided by [46 Page 37]: with $|c|$ the absolute value of the correlation coefficient c: (1) a value for $|c|$ around 0.8 means that the relationship is relatively strong; (2) a value for $|c|$ around 0.3 means a relatively weak relation; and (3) the sign of c gives the direction of the relation. Additionally, we measured the difference between the means of the OO metrics in the presence and absence of anti-patterns using Cliff's *d* (delta), a non-parametric effect size measure [24] for ordinal data. To interpret the difference, we follow the guidelines in [24]: small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$. The procedures for $\mathbf{RQ_1}$ and $\mathbf{RQ_2}$ are described bellow:

- $\mathbf{RQ_1}$: For each class in each JAR, we counted the number of times that it participates in each anti-pattern, and computed OO metrics. We then measured the correlation between these counts and OO metric values by computing Spearman's ρ and Kendall's τ for each pairing. To measure the size effect, we computed Cliff's *d* between OO metric values of classes with a specific anti-pattern and metrics of classes without it.
- $\mathbf{RQ_2}$: As with RQ_1, for each class in each JAR, we counted the number of times that it participates in each code smells, i.e., roles in the anti-patterns. We then measured the correlation between the number of roles played by the class in each anti-pattern and OO metric values. We computed also the effect size between the metric values of classes playing a specific role and that of classes without that role.

For $\mathbf{RQ_3}$ and $\mathbf{RQ_4}$, we used Kruskal-Wallis (always with Bonferroni correction [18]) to determine statistical significance using multiple tests, because the data (occurrences of anti-patterns and the OO metric values) are not drawn from normal distributions and the samples do not have the same sizes. For example, the number of classes with the *AntiSingleton* anti-pattern in Category A is different from Category B, and so on. Therefore, we tested whether the distributions of numbers of anti-patterns (or OO metric values) per category are drawn from the same distributions. The procedures for $\mathbf{RQ_3}$ and $\mathbf{RQ_4}$ are:

- $\mathbf{RQ_3}$: We grouped the classes in each JAR by the category to which they belong. In cases where JARs belonged to multiple categories, we placed their classes in multiple categories. We then used the Kruskal-Wallis test to determine if the frequency distributions of anti-patterns are statistically significant different in the 13 categories (*i.e.*, to determine if a category has a significant effect with an alpha level of 0.05 on the numbers of anti-patterns).
- $\mathbf{RQ_4}$: As with $\mathbf{RQ_3}$, we grouped the classes by category and used the Kruskal-Wallis test to determine if the frequency distributions of the values of a particular metric are significantly different in the 13 categories (*i.e.*, to determine if category has a significant effect with an alpha level of 0.05 on the values of each metric).

4. RESULTS

For the sake of clarity and not to overload the paper with tables, we do not provide tables with all the results. These can be found in our online appendix[4].

4.1 $\mathbf{RQ_1}$: Anti-patterns and Metrics

The results for the relationship between anti-patterns and the values of the OO metrics are summarized in Figure 1 and Table 2. In general, we found only weak correlations except for some outliers. The relationships with metrics are positive and weak, on average; only two anti-patterns (*LazyClass* and *RefusedParentBequest*) exhibit negative and weak correlations with the metrics. The only anti-pattern that exhibits a moderate correlation is *MessageChain*; there is a positive correlation with NOTI (Spearman=0.65, Kendall=0.57). However, according to Cliff's *d*, several metrics have medium-large effects on at least seven anti-patterns: CAM, CBO, CBO-In, CBO-Out, DCC, DSC, LCOM5, NCM, NOM, NOParam, NOTI, RFC, and WMC1. Table 2 lists the top correlated metrics with anti-patterns.

Previous studies [1; 5; 6; 8; 10; 12; 15; 16; 21; 29; 32; 36; 41; 47; 49-52; 55; 57] claim that CK metrics impact quality but we did not found strong correlations with anti-patterns. In the case of CK metrics, none of the correlations with anti-patterns exceed 0.4. The strongest positive relationships are with CBO, RFC, and WMC. The only negative relationship was observed with the

[6] http://wiki.ptidej.net/doku.php?id=sad

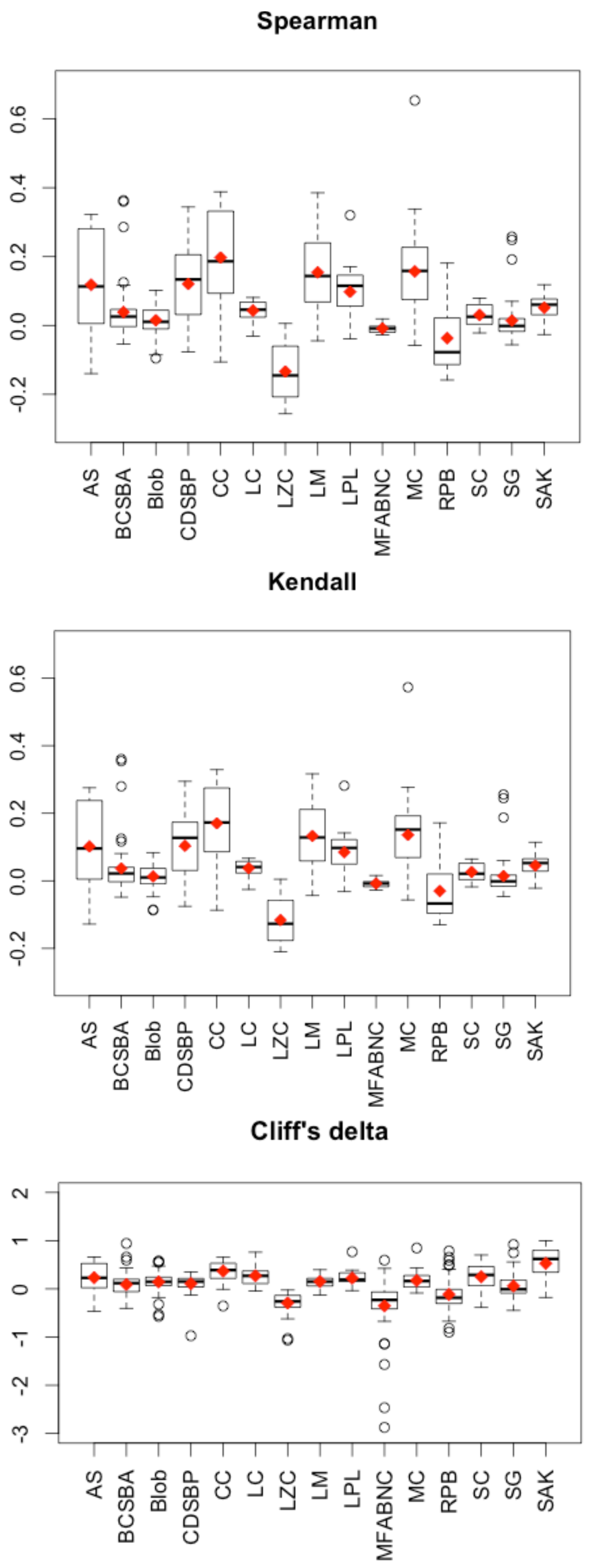

Figure 1. Spearman, Kendall and Cliff's *d* between each anti-pattern and OO metrics.

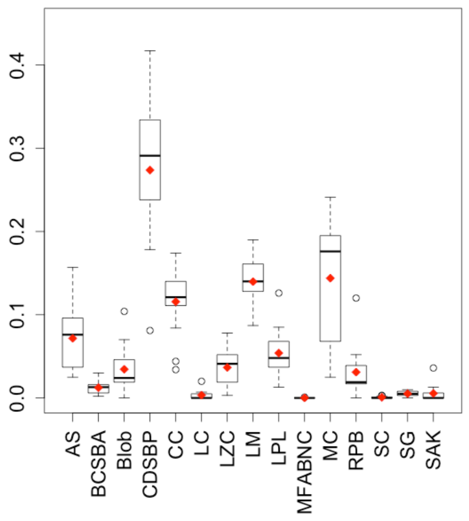

Figure 2. Box-plot of average counts of anti-patterns in ShareJar categories.

metric NOC. For Cliff's *d*, the magnitude of the difference is small for most of the anti-patterns when the metrics are NOC and DIT: for the former (*i.e.*, NOC), we found only a medium difference for *BaseClassShouldBeAbstract*; for the latter we found 5 out of 15 medium/large differences. However, CBO, LCOM5, and RFC have medium-large differences in at least 7 anti-patterns.

Consequently, for **RQ$_1$**, we conclude that, in general, **there are no moderate or strong, only weak, correlations between the classes, in ShareJar applications, participating to anti-patterns and the values of the OO metrics for the classes**. However, we found that the differences between some metrics in classes with some anti-patterns and classes without these anti-patterns are medium and large.

4.2 RQ$_2$: Code Smells and Metrics

There is a positive moderate relationship between the roles of *MessageChain* and NOTI and it is expressed by the two correlation coefficients (*i.e.*, Spearman and Kendall). The rest of relationships between code smells and metrics are low, as in the analysis of the relationship between anti-patterns and metrics. Effect size confirms, at the granularity of roles, the results for **RQ$_1$**. For example, there are positive moderate-large differences in the values of ten metrics in classes with *LargeClass:LargeClassOnly* and 23 metrics in classes with *LargeClass:LowCohessionOnly*. There are negative and large differences in the values of nine metrics in classes with the roles of *ManyFieldAttributesButNotComplex*.

Therefore, for **RQ$_2$**, we conclude that, in general, **there are no moderate or strong correlations between classes, in ShareJar applications, playing roles in anti-patterns and OO metrics**.

4.3 RQ$_3$: Anti-patterns and Categories

Figure 2 depicts the distributions of average occurrences of anti-patterns in each category using box-plots. The differences among the means for each anti-pattern are substantial in several cases; for example, meaningful differences exist even between larger categories with significant overlap; *e.g.*, *AntiSingleton* occurs twice as frequently per class in Category L than in Category D. In

Table 2. Top positive and negative correlated metrics (in bold: moderate, strong correlations; medium, large effect sizes)

Anti-pattern	Spearman	Kendall	Cliff's *d*
AntiSingleton	NAD (0.3219) DAM (-0.1399)	NAD (0.2755) DAM (-0.1280)	**RFC (0.6617)** **MFA(-0.4658)**
BaseClassShould BeAbstract	CLID (0.3632) IR (-0.0539)	CLID (0.3605) DIT (-0.0484)	**CLID (0.9422)** **DIT(-0.4045)**
Blob	DSC (0.1020) DAM (-0.0953)	DSC (0.0837) DAM (-0.0871)	**CBO (0.5727)** **DAM (-0.5723)**
ClassDataShould BePrivate	NAD (0.3441) DAM (-0.3855)	NAD (0.2945) DAM (-0.3527)	**LCOM5 (0.3491)** **DAM(-0.9730)**
ComplexClass	McCabe (0.3883) MFA (-0.1060)	ICHClass (0.3290) MFA (-0.0870)	**RFC (0.6538)** **MFA (-0.3556)**
LargeClass	CBO-Out (0.0813) CA (-0.0309)	CBO-Out (0.0666) CA (-0.0256)	**DSC (0.7603)** CA (-0.0434)
LazyClass	ANA (0.058) CBO-Out(-0.2565)	ANA (0.0047) CBO-Out(-0.2102)	No positive values **CAM (-1.0641)**
LongMethod	LOC (0.3854) NOC (-0.0578)	LOC (0.3167) NOC (-0.0436)	**NOTI (0.964)** MFA(-0.1259)
LongParameter List	NOParam(0.3202) CA (-0.0387)	NOParam (0.2816) CA (-0.0320)	**NOParam(0.7681)** MFA (-0.0366)
ManyFieldAttr ButNotComplex	CBO-In (0.0192) CAM (-0.0274)	CBO-In (0.0158) CAM (-0.0274)	**CBO-In (0.5943)** **CAM (-2.8754)**
MessageChains	**NOTI (0.6536)** NOC (-0.0578)	**NOTI (0.5730)** NOC (-0.0565)	**NOTI (0.8469)** CON(-0.00865)
RefusedParent Bequest	AID (0.1809) WMC (-0.1585)	AID (0.1716) WMC (-0.1297)	**NOH (0.7863)** **CAM (-0.8987)**
SpaghettiCode	WMC (0.0791) MFA (-0.0218)	WMC (0.0648) MFA (-0.0180)	**RFC (0.7023)** **MFA (-0.3872)**
Speculative Generality	CLID (0.2574) CA (-0.0561)	CLID (0.2555) CA (-0.0464)	**CLID (0.9195)** **NOTI (-0.4483)**
SwissArmyKnife	NOP (0.1184) CA (-0.0265)	NOP (0.1143) CA (-0.0219)	**NOP (0.9989)** CON (-0.1785)

general, the distributions are different and the results of Kruskal-Wallis test confirm these differences.

Table 3 lists the *p*-values and the *K* statistic of the Kruskal-Wallis test for **RQ$_3$**. For each anti-pattern (except for *ManyFieldAttributesButNotComplex* that only was detected in Category E), the values of the *K* statistic[7] are greater than the χ^2 values for confidences levels of 0.05 and 0.01 with 12 degrees of freedom $\left(\chi^2_{0.01,12} = 26.2, \chi^2_{0.05,12} = 21\right)$. The differences between the numbers of occurrences of anti-patterns for each category are statistically significant with $\alpha = 0.01$.

ClassDataShouldBePrivate is the most frequent anti-pattern in our dataset (except for Category N—Programming—where the most used is *LongMethod*). *Blob* is the less frequent anti-pattern in Categories M (Emulators) and N (Programming), but not in the others (Figure 3). Again, frequency distributions are different and Kruskal-Wallis test confirms the differences. The values of *K* are greater than the χ^2 values for confidences levels of 0.05 and 0.01 with 12 degrees of freedom $\left(\chi^2_{0.01,12} = 26.2, \chi^2_{0.05,12} = 21\right)$ (except for *ManyFieldAttributesButNotComplex* that only was detected in Category E—Internet). The differences between the numbers of occurrences of anti-patterns for each category are statistically significant with $\alpha = 0.01$.

[7] The *K* statistic is distributed χ^2 with *n*-1 degrees of freedom; in our case *n* is equal to the number of categories, 13.

Table 3. Kruskal-Wallis results for RQ3 with a degree of freedom of 12 (number of categories - 1)

Anti-pattern	*p*-value	*K*
AntiSingleton	0	337.443
BaseClassKnowsDerivedClass	–	–
BaseClassShouldBeAbstract	0	92.464
Blob	0	181.384
ClassDataShouldBePrivate	0	586.629
ComplexClass	0	118.692
FunctionalDecomposition	–	–
LargeClass	0	107.277
LazyClass	0	278.313
LongMethod	0	105.840
LongParameterList	0	366.941
ManyFieldAttributesButNotComplex	0.909	6.139
MessageChains	0	639.969
RefusedParentBequest	0	408.685
SpaghettiCode	0.001	33.148
SpeculativeGenerality	0.001	33.009
SwissArmyKnife	0	156.033
TraditionBreaker	–	–

"–" means that no *p*-value or *K* statistic is available: no occurrences were detected in the categorized applications

Consequently, for **RQ$_3$**, we conclude that in the applications that we analyzed, **there are significant differences among the categories in the frequency distributions of the anti-patterns.**

4.4 RQ$_4$: Categories and Metrics

Table 4 lists the *p*-values and the *K* statistic of the Kruskal-Wallis test for **RQ$_4$**. The results indicate statistically significant differences among the categories for all of the OO metrics for $\alpha = 0.01$ (except for DCAEAC and DCMEC). As with anti-patterns, we infer that meaningful differences exist among the categories in the distribution of the metric values.

Consequently, for **RQ$_4$**, we conclude that in the applications that we analyzed, **there are significant differences among the categories in the distributions of the OO metrics.**

5. ANALYSIS OF THE RESULTS

In this section, we discuss the results of our research questions.

RQ$_1$. The correlation measures (Spearman and Kendall) indicated weak relationships between anti-patterns and OO metrics. However, Cliff's *d* indicated that there are medium and large differences between the values of the OO metrics in classes with anti-patterns and classes without them. To analyze the relationship reported by Cliff's *d*, we counted the number of metrics with medium and large effect size *(*$|d|>=0.33$*)*, grouped by anti-pattern (in Table 5), and the number of anti-patterns in which the metrics have medium and large effect sizes (in Table 6). We computed Cliff's *d* for each metric, comparing metrics of classes participating in particular anti-pattern and metrics of classes not participating in the particular anti-pattern. For example (Table 5), the effect size is large on 16, 19, and 30 metrics for classes with *AntiSingleton*, *ComplexClass*, and *SwissArmyKnife*. In addition, there is a medium difference of CAM and CBO-In (Table 6) between classes with six and five anti-patterns and classes without

Table 4. Kruskal-Wallis results (with Bonferroni correction) for RQ4 with df =12 (number of categories - 1)

Metric	*P*-value	*K*	Metric	*P*-value	*K*
ACAIC	0	43.458	MFA	0	953.141
ACMIC	0	167.931	MOA	0	207.058
AID	0	145.089	McCabe	0	375.512
ANA	0	1228.144	NAD	0	254.852
CA	0	585.445	NADExtended	0	255.024
CAM	0	197.375	NCM	0	961.462
CBO	0	1987.273	NMA	0	183.786
CBO-In	0	2298.643	NMD	0	311.015
CBO-Out	0	1143.478	NMDExtended	0	310.611
CIS	0	395.056	NMI	0	1510.636
CLID	0	68.903	NMO	0	486.851
CON	0	219.648	NOA	0	168.308
DAM	0	661.411	NOC	0	97.217
DCAEAC	**1**	**23.068**	NOD	0	66.633
DCC	0	1987.273	NOH	0	2502.116
DCMEC	**0.336**	**27.415**	NOM	0	311.015
DIT	0	161.278	NOP	0	123.82
DSC	0	4268.073	NOParam	0	477.232
ICHClass	0	267.7	NOTI	0	921.012
IR	0	245.037	NPrM	0	185.899
LCOM1	0	246.594	RFC	0	308.959
LCOM2	0	235.893	SIX	0	302.989
LCOM5	0	315.049	WMC	0	233.292
LOC	0	234.877	WMC1	0	311.015

those anti-patterns. Therefore, although not correlated, some smells do impact negatively quality-related metrics.

We expected many of the results that we obtained. Looking at the top metrics with medium/large effects from anti-patterns (Table 6), we observe that those metrics are related to coupling (CBO, CBO-In, CBO-Out, DCC, and RFC), cohesion (CAM and LCOM5), and design size (DSC, NOParam). The top three metrics in the list (CAM, CBO-In, and LCOM5) have positive and negative effects from anti-patterns. The negative effects are due to the *Blob* (CAM), *LazyClass* (CAM, CBO-In, and LCOM5), *ManyFieldAttributesButNotComplex* (CAM and LCOM5), and *RefusedParentBequest* (CAM and LCOM5) anti-patterns. CAM and LCOM5 measure different aspects of cohesion, because the former is related to similarity of parameter types while the latter to accessing similar fields. These results are not surprising because, by definitions, the methods of classes with the four anti-patterns have methods that do not use the same type of parameters and that do not access similar attributes.

A particular anti-pattern is *LazyClass*. All the medium-large effects on metrics are negative (CAM, CBO-In, DSC, LCOM5, and NOParam). Yet, the values of these metrics are a consequence of the anti-pattern definition, *i.e.*, a "lazy class" does very little and has a minimal definition. Intuitively, the occurrence of *LazyClass* degrades design by making it needlessly cluttered, which consequently may make faults more likely.

Table 5. Numbers of metrics where Cliff's *d* effect sizes are medium and large, listed for each anti-pattern

Anti-pattern	Cliff's *d*				
	(-inf,-0.47]	(-0.47,-0.33]	[0.33, 0.47)	[0.47, inf)	Total
SwissArmyKnife	0	0	1	30	31
ComplexClass	0	1	7	19	27
SpaghettiCode	0	1	16	5	22
AntiSingleton	0	2	2	16	20
ManyFieldAttributesButNotComplex	8	9	2	1	20
LazyClass	4	14	0	0	18
LargeClass	0	0	7	7	14
LongParameterList	0	0	11	1	12
RefusedParentBequest	4	1	3	4	12
MessageChain	0	0	8	1	9
BaseClassShouldBeAbstract	0	3	2	3	8
Blob	1	0	3	3	7
Speculative Generality	0	1	2	2	5
ClassDataShouldBePrivate	1	0	1	0	2
LongMethod	0	0	2	0	2

The top metrics with medium/large effect sizes from the anti-patterns (in Table 6) are CAM, CBO-In, LCOM5, CBO, DCC, DSC, LCOM5, NCM, NMD, NMDExtended, NOM and NOParam, and RFC. These results suggest that metrics for coupling, cohesion, and design size for classes with anti-patterns have medium/large differences with classes without any anti-pattern. Also, the values of some OO metrics, such as coupling metrics, the CK suite, LOC, and McCabe, suggest that classes with some anti-patterns may also be more complex than others. According to other empirical studies [1; 5; 6; 8; 10; 12; 15; 16; 21; 29; 32; 36; 41; 47; 49-52; 55; 57], CBO, RFC, LCOM, and LOC had significant effects on the number of faults in a class and are reliable predictors of fault-proneness. Coupling increases the probability of faults because the stronger the association between classes (inheritance, composition, or method invocations), the stronger the probability of collateral effects when classes are modified or repaired. Classes with high-frequency of outgoing coupling dependencies (out coupling) are more prone to cause surrounding dependent classes to become faulty [9; 13], and modifications or bugs in higher levels of an inheritance tree propagate without control to lower levels. Metrics such as CBO, RFC, and DCC measure those aspects of coupling.

Our results suggest that classes with the *AntiSingleton*, *ComplexClass*, *LargeClass*, *SpaghettiCode*, and *SwissArmyKnife* anti-patterns have greater values of LCOM5 than classes without them. Classes with low cohesion are more likely to be fault-prone, because low cohesion indicates poor design, which is likely to be more fault-prone [10]. The LCOM5 metric measures the extent to which methods use the attributes defined in a class. A value of 1 for LCOM5 indicates that each method of the class references

Table 6. Number of anti-patterns where Cliff's *d* effect sizes are medium and large - listed by metrics.

Metric	**Cliff's *d***				
	(-inf,-0.47]	**(-0.47,-0.33]**	**[0.33,0.47)**	**[0.47, inf)**	**Total**
CAM	4	0	6	0	10
CBO-In	1	0	5	3	9
LCOM5	3	0	1	5	9
CBO	0	1	2	5	8
DCC	0	1	2	5	8
NCM	1	1	3	3	8
NMD	0	2	3	3	8
NMDExt.	0	2	3	3	8
NOM	0	2	3	3	8
NOParam	2	1	3	2	8
RFC	2	1	1	4	8
WMC1	0	2	3	3	8
CBO-Out	0	0	3	4	7
DSC	1	0	4	2	7
NMO	0	2	3	2	7
NOTI	2	1	2	2	7
CIS	0	2	1	3	6
DAM	4	2	0	0	6
MFA	0	4	1	1	6
NOH	0	1	3	2	6
DIT	0	2	2	1	5
NMA	0	1	1	3	5
NOA	1	2	1	1	5
SIX	1	1	2	1	5
AID	0	2	1	1	4
LOC	0	0	1	3	4
McCabe	0	0	1	3	4
NAD	0	0	1	3	4
NADExt.	0	0	1	3	4
WMC	0	0	0	4	4
ANA	1	0	0	2	3
CLID	0	0	1	2	3
ICHClass	0	0	1	2	3
LCOM1	0	0	2	1	3
MOA	0	0	0	3	3
NOD	0	0	2	1	3
NOP	0	1	1	1	3
ACAIC	0	0	2	0	2
ACMIC	0	0	0	2	2
IR	0	0	1	1	2
LCOM2	0	0	1	1	2
NPrM	0	0	1	1	2
DCMEC	0	0	0	1	1
NMI	0	0	0	1	1
NOC	0	0	1	0	1
DCAEAC	0	0	0	0	0
CA	0	0	0	0	0
CON	0	0	0	0	0

only one attribute; a value of zero indicates that each method references every attribute [9]. Thus, classes with values of LCOM5 closer to 1 can be considered as classes that implement several, unrelated functionalities. As a consequence, these classes are more prone to changes and faults; their maintainability is affected by their many responsibilities.

Design-size metrics have been also related to fault-proneness, especially LOC. There is a relationship because the greater the number of lines of code, the greater the probability of introducing a bug during maintenance. Several empirical studies [1; 5; 6; 8; 10; 12; 15; 16; 21; 29; 32; 36; 41; 47; 49-52; 55; 57] provided evidence of a positive relationships between LOC and fault-proneness. Moreover, complex interfaces tend to be more fault-prone [19]. Highly complex interfaces are characterized by a high number of parameters and exit points. Complexity in interfaces introduces complex interaction patterns among classes, because more preconditions must be ensured before invoking methods and developers have more chance to overlook these preconditions. For example, using object types as parameters requires verifying that the parameter values are not null; thus, the longer the list of parameters, the higher the probability to call methods with invalid values.

Thus, from **RQ$_1$**, we infer that anti-patterns generally (though not uniformly) have a relationship with OO metrics, especially with coupling, cohesion, and design size, which suggests a positive relationship with fault-proneness.

RQ$_2$. The relationships expressed in the results for **RQ$_2$** are generally consistent with those that we obtained in **RQ$_1$** without taking roles into account. We found weak correlations between LOC and code smells. However, using Cliff's *d*, we found large differences for the LOC metric values between classes with roles in some anti-patterns (*AntiSingleton*, *ComplexClass*, *SpaghettiCode*, and *SwissArmyKnife*) and classes without those anti-patterns. We believe that our findings have two reasons. First, the presence of these anti-patterns may be a sign of a general poor decomposition of responsibilities, in which fewer, larger classes have too many responsibilities when compared to other classes, i.e., with possible fewer responsibilities and, thus, smaller sizes. Second, our larger dataset may smooth LOC effects because the distribution of LOC values in our dataset with 1,343 applications is different to that from 14 applications used in [44].

Thus, from **RQ$_2$**, we infer that code smells generally (though not uniformly) have a relationship with OO metrics, similarly to anti-patterns, with coupling, cohesion, and design size, which also suggests a positive relationship with fault-proneness.

For **RQ$_3$**, our results show significant differences in the distribution of anti-patterns across categories. While participation in anti-patterns (as expected) is infrequent across all categories (Figure 3), the category to which a class belongs has a substantial impact on the particular anti-patterns that may afflict the classes. This observation answers positively our general question, whether applications in different domains are negatively impacted by smells in different ways. It shows that developers' behavior varies by domain. Developers of certain kinds of applications may find it expedient for domain-related reasons to follow "lazy" conventions, yielding a larger distribution of anti-patterns in their code. These developers may have also less expertise in design and programming than developers in other domains. Developers most likely develop their applications to fulfill needs that no existing applications fulfill, either with respect to functionalities, prices, user-interfaces, availability, licensing, and so on. Thus, developers of applications in the Music category, for example, are more likely to be musicians while developers of applications in the

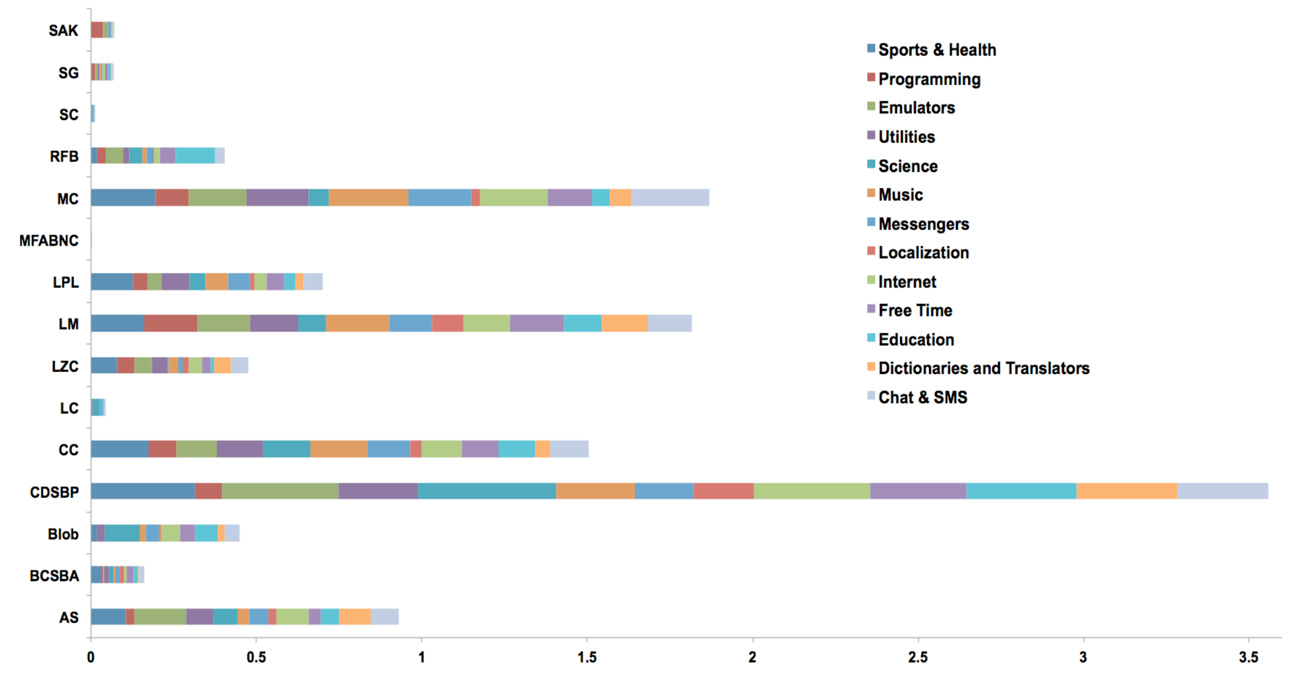

Figure 3. Cumulative average of each anti-pattern per domain (category)

Programming category are more likely to be software developers by profession. Thus, the former developers may be less proficient in design and programming than the latter developers, resulting in lower quality and increased presence of anti-patterns.

A useful direction for future investigation is to study the actual occurrences of anti-patterns in particular domains. Our results indicate that some anti-patterns are more prevalent in some category of applications because of specificities of these applications and observing these categories is useful to warn developers working in such categories about common smells that they may encounter. Even if these smells may be the "best" way to implement these applications because of other constraints on the applications of such categories.

Thus with **RQ$_3$**, we show that applications in different domains are negatively impacted by anti-patterns in different ways and that developers of certain domains, *e.g.*, music, should be wary of smells and–or strive to prevent them, possibly by associating with developers more expert in design and programming (even if less expert in the application domains).

RQ$_4$. Our results, reported in Table 4, show significant differences in the distribution of metric values across categories. This inference is not particularly surprising—developers' discipline and design conventions are intuitively likely to vary by domain—but this deserves more study. One unknown variable is the rationale for this variation. For example, these metrics may vary because of systematic differences in the nature of the problems solved in each domain, but systematic differences in developers' "culture" may also be a factor. This is a question that requires further investigation.

The answer to **RQ$_4$** confirms another intuitive observation from researchers and practitioners: code smells and anti-patterns are not the only factor influencing software quality as we have measured it. Application domain (*i.e.*, the collection of variables associated with it) also has an impact, as do other variables identified in the literature. It is possible that other developers' behaviors associated with the presence of anti-patterns yielded the distribution of metrics that we observed.

Through **RQ$_4$**, we show the relationship between categories and all the computed OO metrics: metric distributions vary across categories with statistical significance and point at differences between developers' discipline and design conventions.

6. THREATS TO VALIDITY

The facts that the studied applications were developed for mobile devices and written uniformly in Java Mobile Edition introduce threats to external validity. Nonetheless, given the size of our dataset, we believe that the results that we obtained from these applications supply a reasonable foundation for future investigation into the impact of anti-patterns with other datasets.

For the sake of simplicity, we computed the metrics, as well as the occurrences of anti-patterns, on the Java bytecode of the mobile applications at the class level. Consequently, any metric that is intrinsically based on the lines of code, such as LOC or WMC, uses the number of bytecodes in the methods of a class as proxy to the number of lines of source code of the class. Although this choice could introduce a bias and be a threat to the construct validity of our study (a LOC would not really represent a line of code), we accept this threat because it is systematic across our complete study and thus does not invalidate comparisons among anti-patterns, domains, and metrics. Compiler optimizations may change the bytecode such that it does not reflect the source code anymore but such changes are intrinsically limited to keep the semantics of the source code and, thus, analyses performed on bytecodes reflect phenomena existing in the source code.

We relied on DECOR for the detection of occurrences of smells. While DECOR has previously obtained 100 percent recall on some datasets and an average precision of 38.2%, its precision and recall cannot be guaranteed in our dataset [25; 40]; accordingly, DECOR constitutes a threat to the construct validity of our study. Its precision values depend on the smells being detected and the applications in which they are detected. Therefore, some reported and studied occurrences of the smells may not be true positive occurrences and may bias the results of our analyses. We mitigate this threat to the construct and conclusion of our study by discussing each role and category independently. We could not mitigate these threats by using versions of the applications because such versions are not currently available. Mäntylä *et al.* [38] report that smells are perceived differently by developers than tools. Therefore, the detected smells may not be relevant to developers and–or in agreement with developers' assessment. We accept this threat to the construct validity of our study because there are no tools to detect developers' perceived smells, and analyzing smells by hand consumes too much time and effort.

We used metrics to assess the quality of the applications with respect to the occurrences (or not) of roles of some smells. We also partly used metrics to identify occurrences of smells, when using DECOR. Therefore, it could be possible that the results observed derived from the use of the same metrics to detect occurrences of smells and to measure quality. However, DECOR does not use only metrics to detect smells but also, among others, the binary-class relationships among classes and their method invocations. Also, a previous paper [33] showed that metrics alone could not explain that a class plays a role in an occurrence of a smell, not even LOC.

The domain and programming language of the applications in our dataset may limit the generalizability of our inferences. Our dataset is comprised only of Java applications for mobile devices; the constraints on these applications (*e.g.*, small memory footprint) presumably do not hold in many other domains, and the occurrences of smells in these applications may vary for practical reasons from that in other systems. However, our goal in this study is exploratory; we expect our study to be replicated on other systems.

In our dataset, 650 out of 1,343 applications are not categorized. However, when investigating the relationships between categories, smells, and metrics (**RQ$_3$, RQ$_4$**), we assumed that the available categories for the applications in ShareJar are comprehensive and that developers "correctly" identified the categories to which their applications belong. Thus, we assumed that 650 applications were not categorized because available categories in ShareJar did not represent the application domains to which the applications belong. Though we believe that this assumption is reasonable, developers are capable of error and repositories are capable of omission; hence, these are threats to validity (external and internal). Again, because ours is an empirical study, we expect these assumptions to be validated via replication on other repositories.

7. CONCLUSIONS

Our study of the relationships among anti-patterns, software categories, and OO metrics shows some evidence that anti-patterns are related to several fault-related metrics. We computed OO metrics on 1,343 Java Mobile Applications because (1) some metrics have known relationships with faults and other quality attributes and (2) they are suitable for computation on a large dataset. We also identified occurrences of code smells and anti-patterns using DECOR [40]. Then, we analyzed the relationship between anti-patterns and metrics using Spearman, Kendall as well as Cliff's d. The results showed that anti-patterns negatively impact software quality-related metrics in Java Mobile Applications, in particular metrics related to fault-proneness. Although we did not find a relationship through correlation analysis, we observed larger differences between metrics of classes, such as coupling, cohesion, and design size, with smells and classes without them. We also observed that anti-patterns have different frequency in the different application domains. Thus, the results of RQ$_1$ through RQ$_4$ and related discussions allow us to answer our general question:

Are applications in different domain negatively impacted by anti-patterns in different ways? **We answer positively this question for anti-patterns. Some anti-patterns are common in all the categories while others are more common in certain application domains.**

Understand the specific reasons for the occurrence of anti-patterns is future work: in some cases, it could be related to specific purposes, such as reducing the application size (*i.e.*, the lower the number of classes, the lower the size of the application) or the memory required by dynamic linking when instantiating classes; in other cases, it could show a lack of developers' knowledge in design concepts. However, independently of the reason, we found large differences between the values of fault-proneness-related metrics (such as coupling, cohesion, and design size) in classes with anti-patterns and classes without them. Thus, anti-patterns in a certain application domains impact negatively quality more than in others. This conclusion suggests a fruitful avenue for future study to determine the domains for which anti-patterns are most harmful. This information would likely be particularly useful to practitioners. Moreover, if systematic differences in quality indeed exist across application domains, it may be useful to develop ways of accounting for these variations, particularly in domains that generally suffer the most from poor quality.

Other avenue involves relating the presence of patterns to fault-proneness *directly*, which (presumably) entails mining substantial quantities of data from software repositories. Another avenue for investigation is also whether anti-patterns are related to readability, which can be assessed using automated techniques (*e.g.*, [14]) and other quality attributes, such as understandability or reusability, that could be measured with dedicated quality models, such as that of Bansiya and Davis [4]. Yet another avenue is the analysis of a larger number of applications, such as ours, while taking into account their evolution and change-proneness (typically computed by differencing versions). In addition, applications written with other technologies, such as Android and iOS apps, should be studied but downloading and decompiling these apps require a major effort when compared to analyzing apps already bundled as JAR files. Finally, the root causes for the presence of more smells in some software categories than in others should be investigated to, possibly, provide means to reduce their prevalence.

8. ACKNOWLEDGEMENTS

This work is supported in part by the NSF CCF-1016868 grant. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

9. REFERENCES

[1] AGGARWAL, K.K., SINGH, Y., KAUR, A., and MALHOTRA, R., 2009. Empirical Analysis for Investigating the Effect of Object-Oriented Metrics on Fault Proneness: A Replicated Case Study. *Software Process: Improvement and Practice 14*, 1 (January), 39-62.

[2] ANTONIOL, G., FIUTEM, R., and CRISTOFORETTI, L., 1998. Design pattern recovery in object-oriented software. In *6th IEEE International Workshop on Program Understanding (IWPC 1998)*, 153-160.

[3] BAKOTA, T., HEGEDUS, P., KÖRTVÉLYESI, P., FERENC, R., and GYIMÓTHY, T., 2011. A Probabilistic Software Quality Model. In *27th IEEE International Conference on Software Maintenance (ICSM'11)*, Williamsburg, Virginia, USA, 243-252.

[4] BANSIYA, J. and DAVIS, C.G., 2002. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering (TSE) 28*, 1 (January), 4-17.

[5] BASILI, V.R., BRIAND, L.C., and MELO, W.L., 1996. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering (TSE) 22*, 10 (October), 751-761.

[6] BINKLEY, A. and SCHACH, S., 1998. Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures. In *20th International Conference on Software Engineering (ICSE'98)*, Kyoto, 452-455.

[7] BRANDT, J., GUO, P.J., LEWENSTEIN, J., KLEMMER, S.R., and DONTCHEVA, M., 2009. Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover. *IEEE Software 26*, 5, 18-24.

[8] BRIAND, L.C., DALY, J.W., PORTER, V., and WÜST, J., 1998. A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems. In *5th International Software Metrics Symposium (METRICS'98)* IEEE Computer Science, Bethesda, MD, 43-53.

[9] BRIAND, L.C., DALY, J.W., and WÜST, J., 1998. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering 3*, 1, 65-117.

[10] BRIAND, L.C., WÜST, J., DALY, J.W., and PORTER, V.D., 2000. Exploring the Relationship between Design Measures and Software Quality in Object-Oriented Systems. *Journal of System and Software (JSS) 51*, 3 (May), 245-273.

[11] BROWN, W.J., MALVEAU, R.C., MCCORMICK III, H.W., and MOWBRAY, T.J., 1998. *AntiPatterns*. John Willey & Sons.

[12] BRUNTINK, M. and VAN DEURSEN, A., 2006. An empirical study into class testability. *Systems and Software 79*, 9 (September), 1219-1232.

[13] BURROWS, R., FERRARI, F., LEMOS, O., GARCIA, A., and TAÏANI, F., 2010. The Impact of Coupling on the Fault-Proneness of Aspect-Oriented Programs: An Empirical Study. In *IEEE 21st International Symposium on Software Reliability Engineering*, San Jose, CA, USA, 329-338.

[14] BUSE, R.P.L. and WEIMER, W.R., 2010. Learning a Metric for Code Readability. *IEEE Transacttions on Software Engineering (TSE) 35*, 4 (July-August 2010), 546-558.

[15] CARTWRIGHT, M. and SHEPPERD, M., 2000. An Empirical Investigacion of an Object-Oriented System. *IEEE Transacttions Software Engineering (TSE) 26*, 7, 786-796.

[16] CHIDAMBER, S., DARCY, D., and KEMERER, C., 1998. Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis. *IEEE Transactions on Software Engineering (TSE) 24*, 8 (August), 629-639.

[17] CHIDAMBER, S.R. and KEMERER, C.F., 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering (TSE) 20*, 6, 476-493.

[18] CORDER, G.W. and FOREMAN, D.I., 2009. *Nonparametric Statistics for Non-Statisticians*. John Wilet and Sons.

[19] DENARO, G., MORASCA, S., and PEZZÈ, M., 2002. Deriving models of software proneness. In *14th international conference on Software engineering and knowledge engineering (SEKE'02)*, Ischia, Italy, 361-368.

[20] DI PENTA, M., CERULO, L., GUÉHÉNEUC, Y.-G., and ANTONIOL, G., 2008. An empirical study of the relationships between design pattern roles and class change proneness. In *24th IEEE International Conference on Software Maintenance (ICSM 2008)*, Beijing, China, 217-226.

[21] EL-EMAM, K., BENLARBI, S., GOEL, N., and RAI, S.N., 2001. The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *IEEE Transactions on Software Engineering (TSE) 27*, 7, 630-650.

[22] FONTANA, F., FERME, V., MARINO, A., WALTER, B., and MARTENKA, P., 2013. Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains. In *IEEE International Conference on Software Maintenance (ICSM '13)*, 260-269.

[23] GAMMA, E., HELM, R., JOHNSON, R., and VLISSIDES, J., 1995. *Design Patterns*. Addison-Wesley Professional.

[24] GRISSOM, R.J. and KIM, J.J., 2005. *Effect Sizes for Research: A Broad Practical Approach*. Lawrence Earlbaum Associates.

[25] GUÉHÉNEUC, Y.-G. and ANTONIOL, G., 2008. DeMIMA: A Multilayered Approach for Design Pattern Identification. *IEEE Transacttions on Software Engineering (TSE) 34*, 5, 667-684.

[26] GUÉHÉNEUC, Y.-G., SAHRAOUI, H., and ZAIDI, F., 2004. Fingerprinting Design Patterns. In *11th Working Conference on Reverse Engineering (WCRE)*, 172-181.

[27] GUÉHÉNEUC, Y.G. and HERVÉ, A.A., 2004. Recovering Binary Class Relationships: Putting Icing on the UML Cake. In *19th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'04)* ACM Press, 301--314.

[28] GUO, Y., SEAMAN, C., ZAZWORKA, N., and SHULL, F., 2010. Domain-specific tailoring of code smells: an empirical study. In *ACM/IEEE 32nd International Conference on Software Engineering (ICSE'10)*, 167-170.

[29] GYIMÓTHY, T., FERENC, R., and SIKET, I., 2005. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Transactions on Software Engineering (TSE) 31*, 10 (October), 897-910.

[30] ISO/IEC, 2001. ISO/IEC 9126. Software Engineering - Product Quality.

[31] JAAFAR, F., GUÉHÉNEUC, Y.-G., HAMEL, S., and F., K., 2013. Mining the Relationship between Anti-patterns Dependencies and Fault-Proneness. In *Working Conference on Reverse Engineering (WCRE'13)*, To appear.

[32] JANES, A., SCOTTO, M., PEDRYCZ, W., RUSSO, B., STEFANOVIC, M., and SUCCI, G., 2006. Identification of defect-prone classes in telecommunication software systems using design metrics. *Information Sciences 177*, 2 (December 15), 3711-3734.

[33] KHOMH, F., DI PENTA, M., GUÉHÉNEUC, Y.-G., and ANTONIOL, G., 2011. An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness. *Empirical Software Engineering (EMSE)*.

[34] KHOMH, F., GUÉHÉNEUC, Y.-G., and ANTONIOL, G., 2009. Playing roles in design patterns: An empirical descriptive and analytic study. In *25th IEEE International Conference on Software Maintenance (ICSM 2009)*, Edmonton, AB, 83-92.

[35] LANZA, M. and MARINESCU, R., 2006. *Object-Oriented Metrics in Practice*. Springer.

[36] LI, W. and HENRY, S., 1993. Object-oriented metrics that predict maintainability. *Journal of Systems and Software 23*, 2, 111-122.

[37] LI, W. and 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.

[38] MÄNTYLÄ, M.V., VANHANEN, J., and LASSENIUS, C., 2004. Bad Smells - Humans as Code Critics. In *International Conference on Software Maintenance (ICSM'04)*, 399-408.

[39] MARINESCU, A., 2004. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In *20th IEEE International Conference on Software Maintenance (ICSM 2004)*, 350-359.

[40] MOHA, N., GUÉHÉNEUC, Y.-G., DUCHIEN, L., and MEUR, A.-F., 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transacttions on Software Engineering (TSE) 36*, 2 (January 2010), 20-26.

[41] OLAGUE, H., ETZKORN, L., GHOLSTON, S., and QUATTLEBAUM, S., 2007. Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes. *IEEE Transactions on Software Engineering (TSE) 33*, 6 (June), 402-419.

[42] OLBRICH, S.M., CRUZES, D.S., and SJOBERG, 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 *IEEE International Conference on Sofftware Maintenance (ICSM'10)*, 1-10.

[43] PALOMBA, F., BAVOTA, G., DI PENTA, M., OLIVETO, R., DE LUCIA, A., and POSHYVANYK, D., 2013. Detecting Bad Smells in Source Code Using Change History Information. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)* (Palo Alto, CA, November 11-15 2013).

[44] POSNETT, D., BIRD, C., and DÉVANBU, P., 2011. An Empirical Study on the Influence of Pattern Roles on Change-Proneness. *Empirical Software Engineering (EMSE) 16*, 3 (June 2011), 396-423.

[45] ROMANO, D., RAILA, P., PINZGER, M., and F., K., 2012. Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes. In *19th Working Conference on Reverse Engineering (WCRE'12)*, 437-446.

[46] ROSS, S., 2009. *Introduction to Probability and Statistics for Engineers and Scientists*. Elsevier Academic Press.

[47] SHATNAWI, R., 2008. The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *Systems and Software 81*, 11 (November), 1868-1882.

[48] SHESKIN, D.D., 2000. *Handbook of Parametric and Nonparametric Statistical Procedures.* Chapman & Hall/CRC.

[49] SINGH, Y., ARVINDER, K., and MALHOTRA, R., 2010. Empirical validation of object-oriented metrics for predicting fault proneness models. *Software Quality 18*, 1, 3-35.

[50] SUBRAMANYAM, R. and KRISHNAN, M.S., 2003. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *IEEE Transactions on Software Engineering (TSE) 29*, 4 (April), 297-310.

[51] SUCCI, G., PEDRYCZ, W., STEFANOVIC, M., and MILLER, J., 2003. Practical assessment of the models for identification of defect-prone classes in object-oriented commercial systems using design metrics. *Systems and Software 65*, 1 (January 15), 1-12.

[52] TANG, M.-H., KAO, M.-H., and CHEN, M.-H., 1999. An empirical study on object-oriented metrics. In *6th International Software Metrics Symposium (METRICS'99)*, 242-249.

[53] YAMASHITA, A. and COUNSELL, S., 2013. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software 86*, 2639-2653.

[54] YAMASHITA, A. and MOONEN, L., 2013. To what extent can maintenance problems be predicted by code smell detection? – An empirical study. *Information and Software Technology 55*, 2223-2242.

[55] YU, P., SYSTA, T., and MULLER, H., 2002. Predicting fault-proneness using OO metrics. An industrial case study. In *6th European Conference on Software Maintenance and Reengineering (CSMR'02)*, 99-107.

[56] ZHANG, C. and BUDGEN, D., 2011. What do we Know about the Effectiveness of Software Design Patterns? *IEEE Transacttions on Software Engineering (TSE) 99*(July).

[57] ZHOU, Y. and LEUNG, H., 2006. Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults. *IEEE Transacttions on Software Engineering (TSE) 32*, 10 (October), 771-789.